People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University M'Hamed BOUGARA – Boumerdes

Institute of Electrical and Electronic Engineering

Department of Electronics

Final Year Project Report Presented in Partial Fulfilment of
the Requirements for the Degree of

MASTER

In **Electronics**

Option: **Computer Engineering**

Title:

IoT Asset Tracking Based on GPS and LoRa Wireless Technology

Presented by:

HAMDI Abdelkhalek

Supervisor:

Pr. KHOUAS Abdelhakim

Registration Number:………/2021

Abstract

Asset tracking is the process of collecting location data regarding valuable items within an organization. The Internet of Things (IoT) stands for connecting physical objects wirelessly, for the purpose of exchanging data over the internet. IoT asset tracking defines the use of IoT enabling technologies in order to collect, store, and visualize location data of assets in real-time, which allows better and more reliable decision making. Technologies such as Wi-Fi and cellular are widely used to enable IoT wireless connection of physical objects; however, when dealing with use cases in large industrial environments where assets are spread randomly indoor and outdoor, the network coverage problem arises. This report proposes an IoT solution to the asset tracking use case in which Global Positioning System (GPS) is used to acquire real-time location data and LoRa communication technology is used to allow wireless data transmission over long ranges. A tracker and a gateway embedded systems were designed and implemented based on microcontrollers and System-on-Chip (SoC) modules. The tracker was designed to acquire and send location data wirelessly over long ranges; the gateway on the other hand receives data then forwards it to a web application for storing and visualization. Different software drivers were written using C programming language to allow interfacing the GPS and LoRa modules. By the end of this report, we were able to visualize real-time locations of a moving vehicle with 1-3 meters accuracy in localizing the asset, and a wireless signal reach of 293 meters. Our solution is used to track outdoor moving assets within industrial environments.

Dedication

“To my beloved mother and father.”

∞ *Abdelkhalek* ∞

Acknowledgment

First and foremost, I would like to praise and thank God, the almighty, who has granted countless blessings, knowledge, and opportunity that allowed me to complete this work. Secondly, I would like to express my deepest appreciation to all institute teachers for their help during the whole five academic years. A special gratitude I give to my final year project supervisor, Pr. Abdelhakim Khouas, whose contribution in stimulating suggestions and encouragement, helped me to realize my project and to complete this report. My appreciation is also extended to Mr. Christian Baumgaertel who was my supervisor during my internship at Roche Diagnostics in Germany, and who had the credit to introduce me to the techniques that I used in this report.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ADC	Analog-to-Digital
ASIC	Application-Specific Integrated Circuits
BLE	Bluetooth Low-Energy
BW	Band Width
CSS	Chirp Spread Spectrum
CR	Code Rate
CPU	Central Processing Unit
CF	Carrier Frequency
CRC	Cyclic Redundancy Check
COTS	Commercial Off-the-Shelf
DAC	Digital-to-Analog
DBMS	Database Management System
DR	Data Rate
FPGA	Field Programmable Gate Arrays
FEC	Forward Error Correction
FIFO	First In First Out
GNSS	Global Navigation Satellite System
GNU	GNU's Not Unix
GPS	Global Positioning System
Hz	Hertz
IIoT	Industrial Internet of Things
IoT	Internet of Things
IO	Input/Output
I^2C	Inter-Integrated Circuit
IP	Internet Protocol
IEEE	Institute of Electrical and Electronics Engineers
IDE	Integrated Development Environment
LPWAN	Low-Power Wide-Area Network
LoRa	Long Range
M2M	Machine to Machine
MCU	Microcontroller
MISO	Master-In Slave-Out
MOSI	Master-Out Slave-In

MIPS	Microprocessor without Interlocked Pipelined Stages
mW	Milliwatts
NFC	Near field communication
NB-IoT	Narrow-Band Internet of Things
OSI	Open Systems Interconnection
OOP	Object-Oriented Programming
PNT	Positioning, Navigation and Timing
QR	Quick Response
RTOS	Real-Time Operating Systems
RFID	Radio Frequency Identification
RF	Radio Frequency
RAM	Random Access Memory
SDK	Software Development Kit
SF	Spreading Factor
SoC	System-on-Chip
SPI	Serial Peripheral Interface
TP	Transmission Power
TSDB	Time Series Database
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver and Transmitter
WSN	Wireless Sensor Networks

Introduction

The Internet of Things, abbreviated IoT, is the new approach of connecting everything to the internet for the purpose of monitoring, controlling, and digitalizing physical objects to minimize human-to-human or human-to-machine interaction. For data acquisition, IoT uses mainly physical sensors to capture data from any process or environment, this data is collected then sent to centralized hubs for further analysis and processing. IoT sensors are built on top of embedded systems and they mostly provide wireless communication for data transmission. The Industrial Internet of Things (IIoT) refers to the extension and use of IoT in industrial sectors and applications. With a strong focus on machine-to-machine (M2M) communication, IIoT enables industries to have better efficiency and reliability in their operations. IoT incorporates thousands of industrial applications, among which IoT asset tracking is a fundamental use case that is highly needed due to the precious information that it provides regarding different processes within industrial environments. Asset tracking provides reliable and efficient monitoring of physical assets, thus enabling users to keep close track of items as they move throughout the workplace and between sites which increasingly enables better decision-making. IoT asset tracking refers to the method of localizing physical assets in real-time, either by scanning barcode labels attached to the assets or by using other localization techniques such as Global Positioning System (GPS). Location data is broadcast to the user through wireless communication technologies like Wi-Fi. When it comes to wireless data communication in industrial environments, a set of constraints arise as these environments are not considered to host such technologies. In large industries where processes are distributed everywhere indoor and outdoor, it is hard to cover most of the processes with wireless communication technologies such as Wi-Fi and Bluetooth. Furthermore, if one of the Cellular technologies like 4G is to be used to connect a considerable amount of objects that may reach thousands in a single firm, the price becomes significantly expensive.

This work introduces a cost-effective solution to the IoT asset tracking use case, the aim of which is to design and implement embedded systems that allow outdoor mobile assets localization within industrial environments. The project is based on a long-range wireless communication approach for the reason to solve the issue of network coverage; it consists of three main parts: a tracker node which can be attached to mobile assets for localization, a network-connected gateway to receive data from trackers over long ranges, and a web application which allows storing and visualizing location data received from the gateway in an interactive graphical map. The advantage of such an approach appears from the network coverage perspective, as only the gateway should have Internet access, meanwhile, the rest of the tracker nodes can move freely and still be able to send location data wirelessly over long ranges to the gateway.

The report is structured as follow: chapter one provides a global overview of IoT, wireless communication, and asset tracking use case; the theoretical background with all needed techniques is explained in chapter two; chapter three contains a detailed design and implementation steps of the asset tracking solution; chapter four illustrates and discusses the acquired results. At the end of the report, a general conclusion is added to summarize and provide suggestions for further work.

Chapter 1: IoT and Asset Tracking

1.1. The Internet of Things

IoT defines a connected network of physical objects with embedded sensors and actuators, which can communicate through a wired or wireless communication network. IoT includes the process of sensing environmental data from one side and interacting with the physical world (control/actuation) from another side with minimal human intervention; this logic is built on top of the existing communication technologies. J. Gubbi et al. stated in [1] that from a top-level perspective, there exit three components included in most IoT systems: Hardware—made up of sensors, actuators, and embedded communication hardware; Middleware—on-demand storage and computing tools for data analysis; Presentation—easy to understand visualization and interpretation tools that can be widely accessed on different platforms and which can be designed for different applications. Figure 1.1 illustrates a top-level overview of the IoT components along with data path between different blocks.

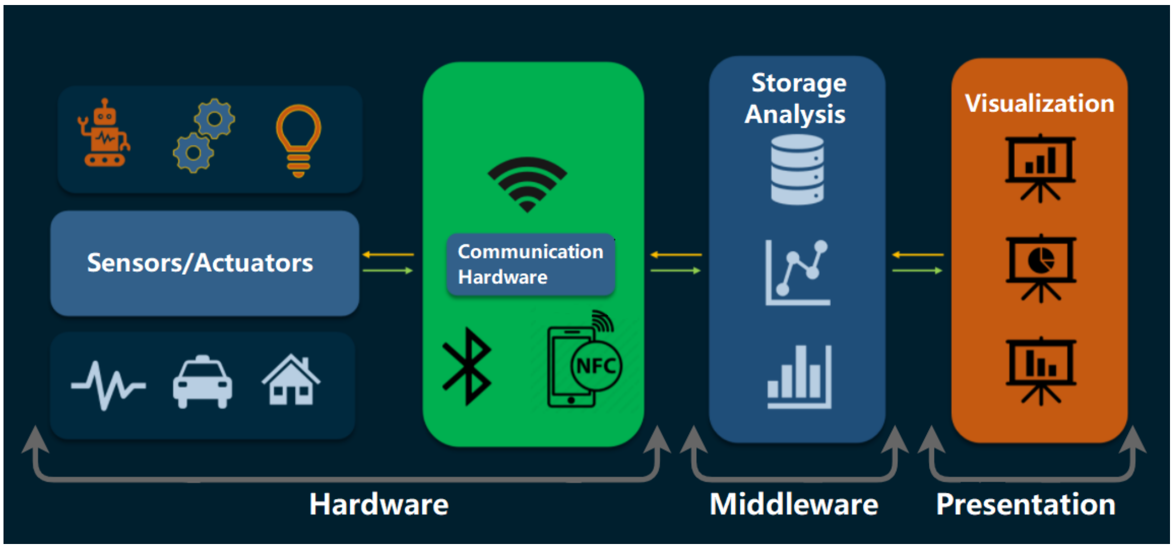


Figure 1.1: A top level overview of IoT components.

IoT incorporates a huge number of connected objects such that for large scaling purposes, different technologies and approaches have been developed to enable successful IoT devices deployment, among which the following are widely used nowadays [1]:

1) *Wireless Sensor Networks (WSN)*

Recent advances in wireless communications and digital electronics have enabled the development of low-cost, low-power, multifunctional sensor nodes that can communicate upon short/long distances. A sensor node consists of sensing, data processing, and communicating components. Sensor data are shared among sensor nodes and sent to a distributed or centralized system for analysis [2].

2) Addressing schemes

The ability to uniquely identify 'Things' is critical for the success of IoT. This allows to uniquely identify billions of devices and to control remote devices through the Internet. The few most critical features of creating a valid address are: uniqueness, reliability, persistence, and scalability [1].

3) Data storage and analysis

As a result of the massive spread of IoT-enabled devices, a huge amount of data is sourced every second, this leads to critical challenges on how to store and use this data in the most efficient way. The data have to be stored and used intelligently for smart monitoring and actuation in conjunction with intelligent algorithms that make sense of the data collected in order to achieve automated decision-making [1].

4) Visualization

Data visualization is the graphical representation of information and data. By using visual elements like charts, graphs, and maps; data visualization tools provide an accessible way to see and understand trends, outliers, and patterns in data [3]. Visualization in IoT applications allows the interaction of the end-user with the environment in an intuitive and easy-to-understand way and helps to perform effective and immediate decision-making.

The term "Internet of Things" was first introduced by Kevin Ashton in 1999 in the context of supply chain management [4]. However, in the last few years, this definition has covered a wide range of applications like healthcare, agriculture, transport, etc. From production line and warehousing to retail delivery and store shelving, IoT is transforming business processes by providing more accurate and real-time visibility into the flow of materials and products. Firms will invest in IoT to redesign factory workflows, improve tracking of materials, and optimize distribution costs. For example, some shipping companies are already using IoT-enabled fleet tracking technologies to cut costs and improve supply efficiency. In industry, IoT monitoring and control systems collect data about equipment performance, energy usage, and environmental conditions, which allow end-users to constantly track the overall performance of different processes in real-time. The resulting data allow information sharing and collaboration that enhance situational awareness and avoid information delay and distortion [5]. Figure 1.2 shows the major application domains tackled by IoT.

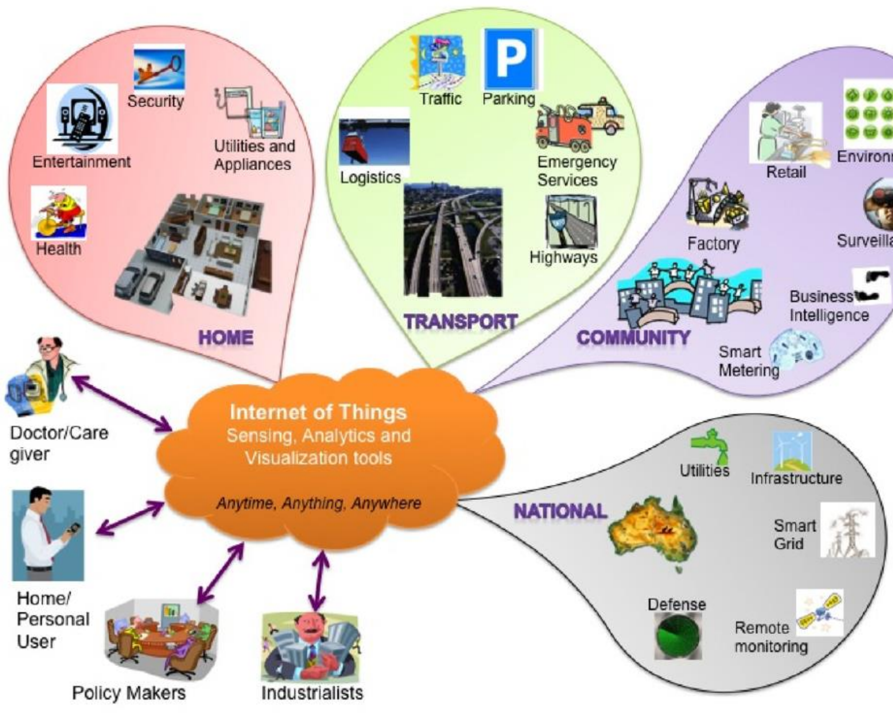


Figure 1.2: Internet of Things major applications [1].

1.2. IoT wireless technologies

Connectivity is the key enabler for successful IoT deployment. However, with innumerable IoT use cases and applications, multiple communication solutions vary depending on every use case. Each communication technology is characterized in terms of range, scalability, cost, and network requirements as presented in Figure 1.3; this provides a wide range of choices for developers and end-users and covers most of application fields and their scenarios as shown in Table 1.1. Albert Behr introduces in [6] the leading wireless communication technologies used in IoT and their applications, these are summarized in subsections bellow:

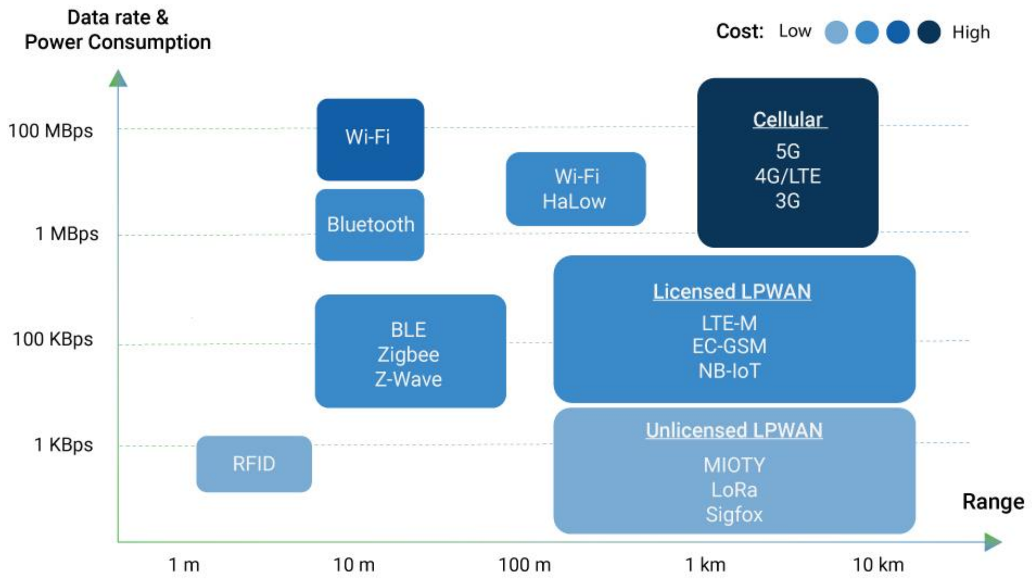
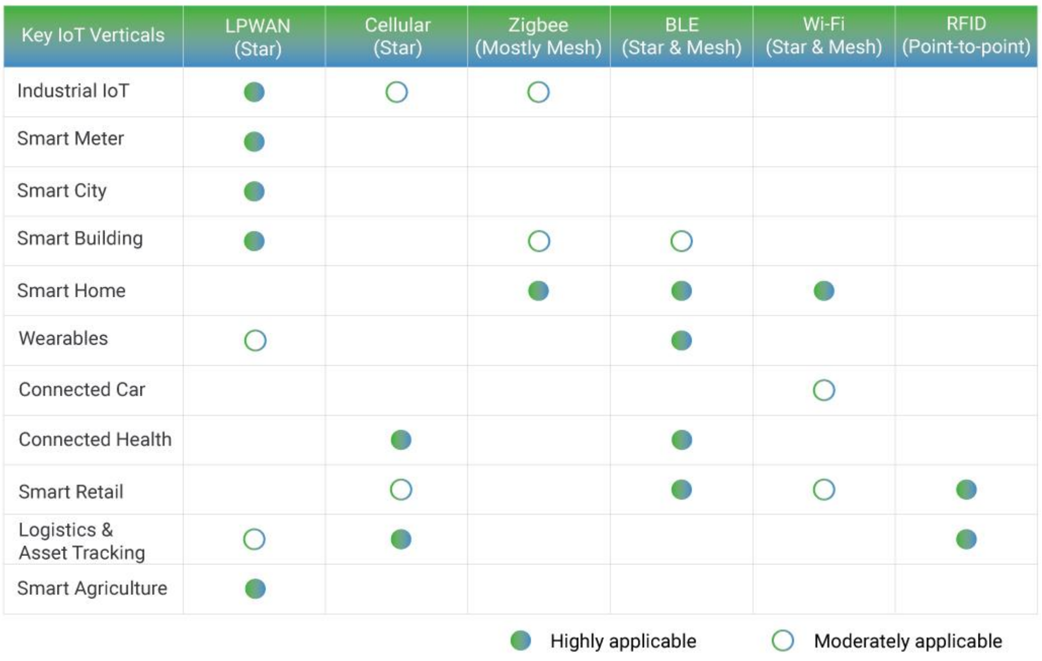


Figure 1.3: Wireless technologies comparison in terms of range, data rate, and cost [6].

Table 1.1: IoT applications vs wireless technologies [6].



Key IoT Verticals	LPWAN (Star)	Cellular (Star)	Zigbee (Mostly Mesh)	BLE (Star & Mesh)	Wi-Fi (Star & Mesh)	RFID (Point-to-point)
Industrial IoT	●	○	○			
Smart Meter	●					
Smart City	●					
Smart Building	●		○	○		
Smart Home			●	●	●	
Wearables	○			●		
Connected Car					○	
Connected Health		●		●		
Smart Retail		○		●	○	●
Logistics & Asset Tracking	○	●				●
Smart Agriculture	●					

● Highly applicable ○ Moderately applicable

1.2.1. LPWAN

Low-Power Wide-Area Network (LPWAN) is a new phenomenon in IoT. Providing long-range communication on small, inexpensive batteries that last for years, this technology best fits large-scale industrial IoT applications. LPWANs can connect several types of IoT sensors and facilitate numerous applications from remote monitoring and worker safety to building controls and facility management. However, LPWANs can only send small blocks of data at a low rate, and therefore are better suited for use cases that don't require high bandwidth and are not time-sensitive.

1.2.2. Cellular (3G/4G/5G)

Cellular networks offer reliable broadband communication for voice and video streaming applications. On the downside, they impose very high operational costs and power requirements. While cellular networks are not viable for the majority of battery-operated IoT devices, they fit well in use cases such as connected cars and fleet management.

1.2.3. Zigbee and Other Mesh Protocols

Zigbee is a short-range, low-power, wireless standard (IEEE 802.15.4), commonly deployed in a mesh topology to extend coverage by relaying sensor data over multiple sensor nodes. Compared to LPWAN, Zigbee provides higher data rates, but at the same time, much less power-efficiency due to mesh configuration. This technology is best-suited for medium-range IoT applications with an even distribution of nodes in close range.

1.2.4. Bluetooth and BLE

Bluetooth is a short-range wireless technology standard used for exchanging data between fixed and mobile devices over short distances. The new Bluetooth Low-Energy (BLE), also known as Bluetooth Smart, is further optimized for IoT applications with low power capabilities. In retail contexts, BLE can be coupled with beacon technology for enhanced customer services like in-store navigation, personalized promotions, and content delivery.

1.2.5. Wi-Fi

Except for few applications like digital signage and indoor security cameras, Wi-Fi is not often a feasible solution for connecting IoT devices because of its major limitations in coverage, scalability, and power consumption. Instead, the technology can perform as a back-end network for offloading aggregated data from a central IoT hub to the cloud, especially in smart home applications. Critical security issues often prevent its adoption in industrial and commercial use cases.

1.2.6. RFID

Radio Frequency Identification (RFID) uses radio waves to transmit small amounts of data from an RFID tag to a reader within a very short distance. Until now, this technology has promoted a major revolution in retail and logistics. By attaching an RFID tag to various products and equipment, businesses can track inventory and assets in real-time enabling better stock and production planning as well as optimized supply chain management. Alongside increasing Industrial IoT adoption, RFID continues to be established in the retail sector, enabling applications like smart shelves and self-checkout.

1.3. Asset tracking technologies and applications

An asset is an item, thing or entity that has potential or actual value to an organization [7], this includes equipment, stock, vehicles, plant and machinery etc. For all asset-intensive businesses that rely heavily on equipment to generate revenue, effective asset tracking is crucial. Traditionally, the purpose of asset tracking was to answer the question: “how much of (something) do I have?” but, with the huge advance in technology and digital transformation, more meaningful data can be acquired to give better insights into the status, location, maintenance schedule, and other information related to physical assets. Modern tracking technologies come in the form of Asset Tracking Software, which digitalizes data into one easy-to-access database; also known as a fixed asset register. A fixed register consists of all types of data including purchase costs, real-time locations and status, location history, user history, and maintenance history [8]. Missing equipment and improperly maintained equipment can directly affect business profit, thus, finding and replacing any lost or missing physical assets is a keen responsibility. Taking an example of constantly moving vehicle like a forklift, tracking such an asset provides overall visibility of the current fleet, not only by knowing the exact location of each vehicle in real-time but also by having the ability to track their historical locations, distance traveled by each forklift, fuel volume, current speed, run time versus stop time, the ratio of loaded and unloaded time, the time needed to complete a task and even more. Given these insights, logistic managers can improve processes and the overall efficiency and utilization of their fleet.

Tracking physical assets in different environments is a challenging problem. A wide range of tracking technologies is available to cover all use cases, thus, choosing the right technology for a specific application depends mainly on the environment where assets are deployed. There exist three basic technology types that can be used for localization [10]:

- **Presence-Based systems:** the location of a reading device is predefined, so whenever the reader detects a tag it gives the information that the object is present at this point. It makes no further measurement about the exact location of that tag. Examples of presence-based systems are RFID and Bluetooth tags.
- **Proximity-Based systems**: similar to presence-based however the reader can determine how close a tag is to the predefined location based on the received power. Examples of proximity-based systems are Wi-Fi and Bluetooth beacons.
- **Time-Based systems**: they are more complex and rely on the measurement of time and thus distance between the tag and the reader based on the transmitted signals. GPS for example is a time-based localization system.

1.3.1. Outdoor asset tracking

Outdoor tracking deals with assets that operate in open spaces like vehicles. Businesses such as construction, gas and oil, logistics and delivery, implement outdoor positioning for their assets due to the valuable information that it provides. Satellite-based positioning technology - Global Navigation Satellite System (GNSS) and GPS - is most commonly used for outdoor asset tracking. Essentially, it employs receivers to collect radio signals that are transmitted through a collection of satellites that circle the Earth, these signals are used to accurately calculate and identify an outdoor location that is expressed as latitude and longitude [11].

1.3.2. Indoor asset tracking

Indoor tracking is a technique that provides a continuous and real-time location of objects within a closed space. It is primarily used in retail floors, warehouses, factories, and offices to monitor and track people, equipment, and merchandise. Unlike outdoor positioning systems where 1-3 meter accuracies can be achieved using GNSS, indoor positioning cannot effectively use GNSS for location detection [12]; instead, indoor positions are calculated using the information transmitted by mobile tags using technologies like RFID Tags, QR Codes, NFC Tags, Bluetooth beacons, … etc.

Chapter 2: Theoretical Background

2.1. Global Positioning System

GPS is a U.S.-owned utility that provides users with positioning, navigation, and timing services (PNT services). This system consists of three segments: the space segment, the control segment, and the user segment [13]. The space segment consists of a constellation of satellites transmitting radio signals to users. GPS satellites fly in medium Earth orbit at an altitude of approximately 20,200 km. Initially, satellites were arranged into six equally-spaced orbital planes surrounding the Earth. Each plane contains four "slots" occupied by baseline satellites. This 24-slot arrangement ensures users can view at least four satellites from virtually any point on the planet. As of January 9, 2021, there were a total of 31 operational satellites in the GPS constellation on-orbit spares [13].

The control segment consists of a global network of ground facilities that track the GPS satellites, monitor their transmissions, perform analyses, and send commands and data to the constellation. *Monitor Stations* track GPS satellites as they pass overhead, collect navigation signals, range/carrier measurements, and atmospheric data then feed observations to the master control station. *Master Control Station* provides command and control of the GPS constellation, uses global monitor station data to compute the precise locations of the satellites, and generates navigation messages for upload to the satellites. *Ground Antennas* send commands, navigation data uploads, and processor program loads to the satellites [13].

The user segment consists of the GPS receiver equipment, which receives the signals from the satellites and uses the transmitted information to calculate the user's three-dimensional position and time [13]. To calculate location, a GPS device must be able to read the signal from at least four satellites. The receiver uses the signals it receives to determine the transit time of each message and computes the distance to each satellite using the velocity of light, the measured distance is called Pseudorange. Figure 2.1 illustrates how location is determined based on four satellite readings.

2.1.1. NMEA Standard

NMEA is an acronym for the National Marine Electronics Association. NMEA existed well before GPS was invented. According to the NMEA website [14], the association was formed in 1957 by a group of electronic dealers to create better communications with manufacturers. Today in the world of GPS, NMEA is a standard data format supported by all GPS manufacturers, much like ASCII is the standard for digital computer characters in the computer world. NMEA standard uses a simple serial communications protocol that defines how data is transmitted in a "sentence" from one "talker" to multiple "listeners" at a time, this data can be transmitted via different types of communications interfaces such as RS-232, USB, Bluetooth, Wi-Fi, and many others. Most computer programs that provide real-time position information understand and expect data in the NMEA format, this data includes the complete PVT (position, velocity, time) solution computed by the GPS receiver [15].

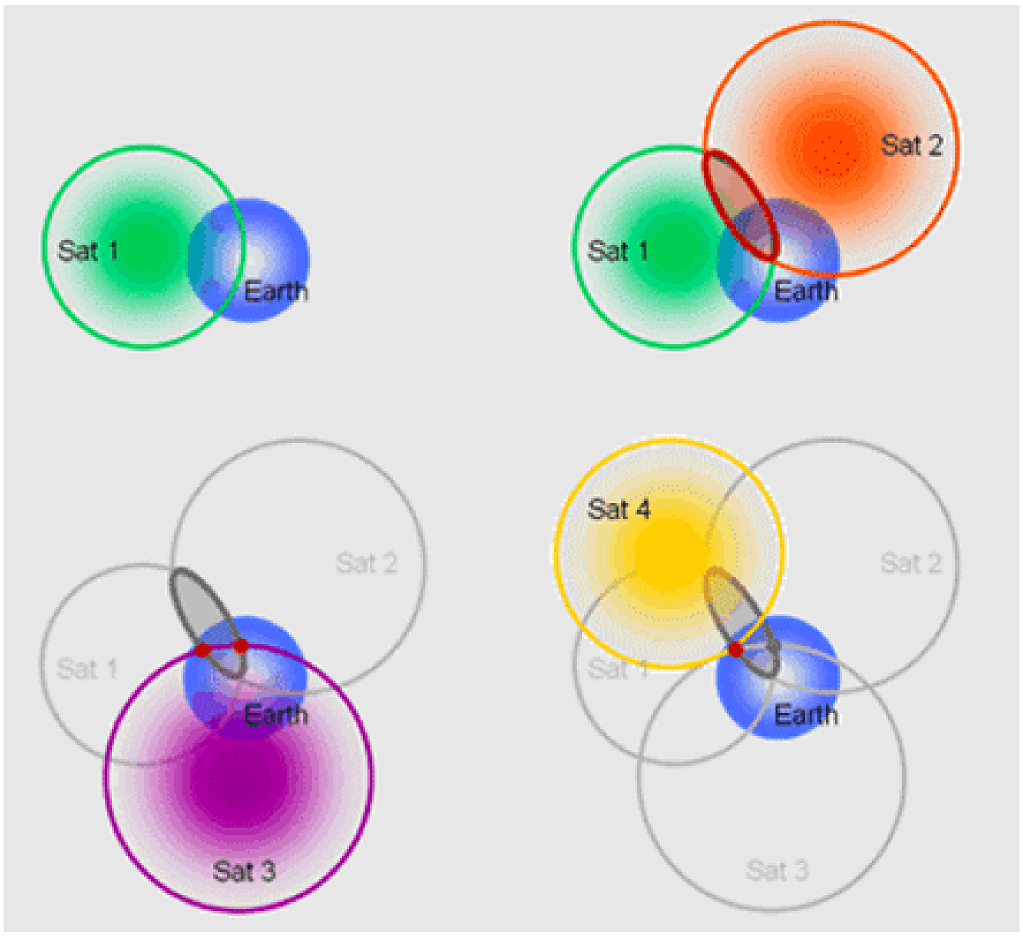
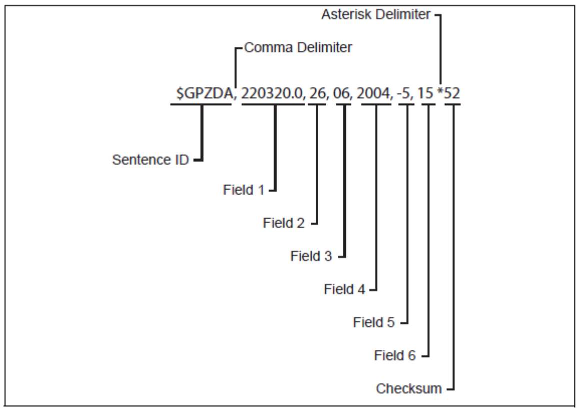


Figure 2.1: GPS localization illustration using four satellites [13].

The idea of NMEA is to send a line of data called a sentence that is self-contained and independent from other sentences. There are standard sentences for each device category and there is also the ability to define proprietary sentences for use by manufacturing companies. Each NMEA sentence includes an identifier to distinguish it from other messages in the data stream, one or more fields of data separated by a comma, and a checksum preceded by an asterisk symbol (*) to validate the data. NMEA messages always begin with a dollar sign ($) followed by a talker ID code, for example “GP” for GPS devices, and a message ID code, for example “GGA” for GPS Data Fix. Figure 2.2 shows an example of the “ZDA” NMEA sentence [16].



Field	Data	Description
1	220320.0	Time in UTC
2	26	Day (01 to 31)
3	06	Month (01 to 12)
4	2004	Year
5	-5	Local time zone offset from GMT (in hours 00 to ±13 hours)
6	15	Local time zone offset from GMT (in minutes)

Figure 2.2: Example of the "ZDA" NMEA sentence with its field's description [16].

2.2. LPWAN and LoRa

LPWAN is a type of wireless communication that is designed for sending small data packages such as sensor data over long distances. There are several competing technologies in the LPWAN space from which Narrowband IoT (NB-IoT), Sigfox, LoRa are widely used. LPWAN Network is an emerging network technology for IoT that offers long-range and wide-area communication at low power. In contrast to traditional short-range wireless sensor networks, the design goal of LPWANs is to offer wide-area coverage at low power, and low cost. Most non-cellular LPWANs operate on low frequencies (sub-GHz band) that provide a long communication range, from few kilometers in urban areas to tens of kilometers in rural areas. Lower frequencies have better propagation characteristics through obstacles. These properties made the sub-GHz band attractive for LPWANs technologies [18].

IoT devices are expected to operate for a very long time (several years) without the need for battery replacement. LPWANs achieve low-power operation through the use of the star topology, which eliminates the energy consumed through packet routing in multihop networks. Second, they keep the node design simple by offloading the complexities to the gateway, as they use narrowband channels to decrease the noise level and extend the transmission range [18]. A major factor that contributed to the rise of LPWANs is its low cost. Non-cellular LPWANs require no (or limited) infrastructure and operate on an unlicensed spectrum, providing an excellent alternative to the cellular network. In addition, the advances in the hardware design and the simplicity of LPWAN end-devices makes LPWANs economically viable [19]. LPWANs are designed to provide reliable and robust communications. Most LPWANs adopt robust modulation techniques and spread-spectrum techniques to increase the signal resistance to interference and provide a level of security. In spread-spectrum, the narrowband signal is spread in the frequency domain with the same power density resulting in a wider bandwidth signal [20].

2.2.1. What is LoRa?

Lora is a wireless technology that offers long-range, low-power, and secure data transmission for M2M and IoT applications. Technically, it is a radio modulation scheme; that is a way of manipulating radio waves to encode information using a technique derived from *Chirp Spread Spectrum* modulation [21]. According to the OSI seven-layer Network Model, Lora can be considered as a physical layer implementation (PHY) or “bits” layer implementation, depicted in Figure 2.3, thus, it can be integrated along with higher-layer implementations which allows it to coexist and interoperate with the existing network architectures. Instead of cabling, the air is used as a medium for transporting LoRa radio waves from an RF transmitter in an IoT end node to an RF receiver in a gateway, and vice versa [22]. The LoRa wireless technology was developed by a French start-up called Cycleo which developed the LoRa modulation technology. In 2012, the Semtech Corporation acquired Cycleo. The LoRa radio and modulation part is patented, and its source is closed. Semtech has licensed its LoRa intellectual property (IP) to other chip manufacturers, such as HopeRF, Microchip, Dorji, etc. The word LoRa is a trademark of Semtech Corporation, filed in 2015 [23].

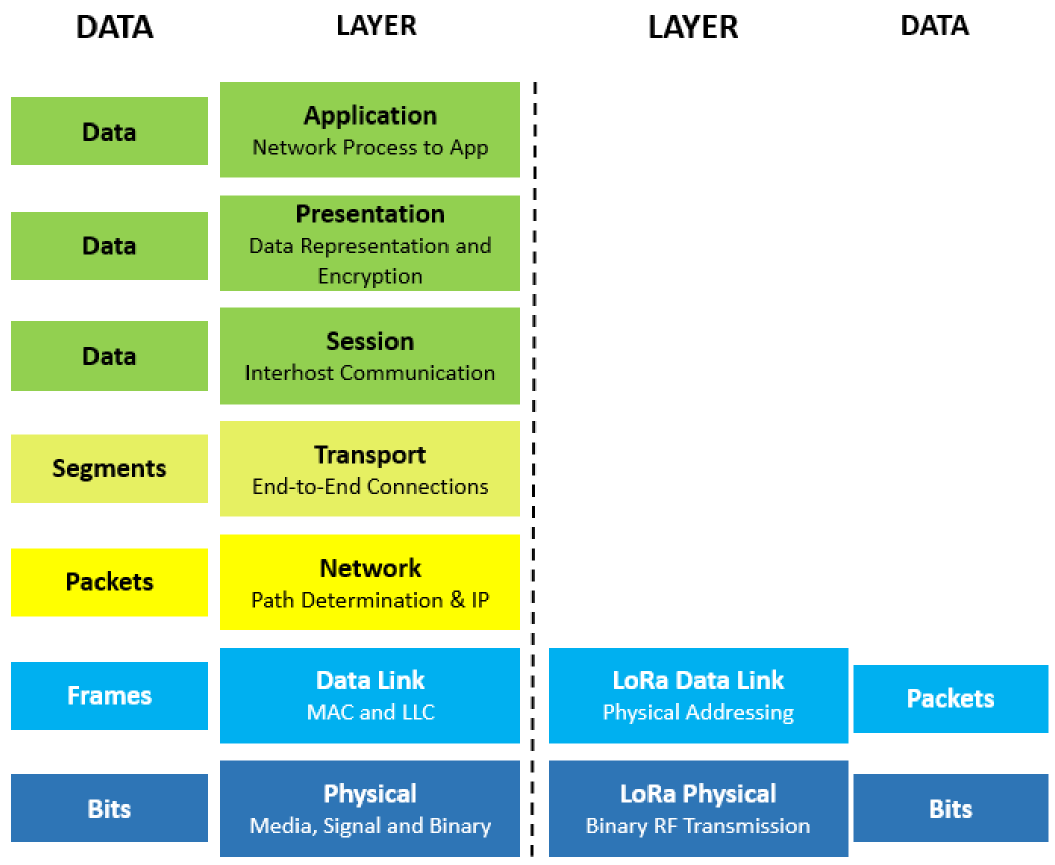


Figure 2.3: OSI seven-layer network model [23].

LoRa end nodes are responsible for collecting data from the environment and send it via RF waves to a LoRa gateway. LoRa gateways act as an intermediary between LoRa end nodes and cloud services, meaning that a gateway is a hub where all data is gathered, converted into network packets, and then forwarded to a web application via cellular, Ethernet, or Wi-Fi to make use of it. The data may be cleaned, stored, visualized, analyzed, and decisions may be sent back as an action to a node or an end-user based on the acquired data. Figure 2.4 shows the basic architecture of both LoRa gateway and end node.

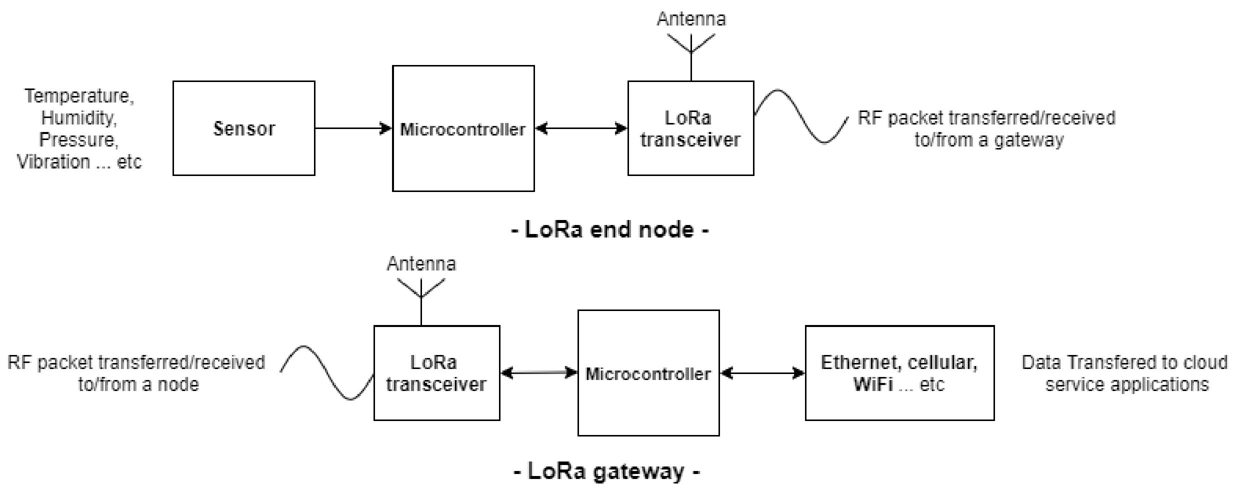

Antenna
Temperature, Humidity, Pressure, Vibration ... etc
Sensor
Microcontroller
LoRa transceiver
RF packet transferred/received to/from a gateway
- LoRa end node -
Antenna
RF packet transferred/received to/from a node
LoRa transceiver
Microcontroller
Ethernet, cellular, WiFi ... etc
Data Transfered to cloud service applications
- LoRa gateway -

Figure 2.4: LoRa gateway and end node architectures.

2.2.2. LoRa characteristics

A key characteristic of the LoRa-based solutions is ultra-low power requirements, which allows for the creation of battery-operated devices that can last for up to 10 years. Deployed in a star topology, LoRa is perfect for applications that require long-range or deep in-building communication among a large number of devices that have low power requirements and that collect small amounts of data [22]. Concerning range, a single LoRa-based gateway can receive and transmit signals over a distance of more than 15 kilometers in rural areas. Even in dense urban environments, messages are able to travel up to three to five kilometers, depending on how deep indoors the end nodes are located. Table 2.1 shows approximately the ranges that LoRa can reach in different environments [22]. As far as battery life is discussed, the energy required to transmit a data packet is quite minimal given that the data packets are very small with respect to the transmission rate. Furthermore, when the end devices are asleep, the power consumption is measured in milliwatts (mW), allowing a device’s battery to last for many years [22]. Given the capabilities of LoRa-based end nodes and gateways, only a few gateways configured in a star network are required to serve a huge number of end nodes, this means that expenses can be kept relatively low. Also, when cheap LoRa RF modules are embedded in inexpensive end nodes and are used in conjunction with the open protocols standard, the return on investment can be considerable [22].

Table 2.1 : LoRa approximate ranges [25].

Environment	Range (km)
Urban areas (towns & cities)	2-5
Rural areas (countryside)	5-15
Direct Line Of Sight	>15

2.2.3. LoRa modulation technique

In digital communications, chirp spread spectrum (CSS) is a spread spectrum technique that uses wideband linear frequency modulated chirp pulses to encode information. CSS uses its entire allocated bandwidth to broadcast a signal, making it robust to channel noise [24]. A chirp is a sinusoidal signal whose frequency increases or decreases over time to represent a piece of information; if the frequency increases, it is called an up-chirp, whereas a decreasing frequency chirp is called a down-chirp [24]. Figure 2.5 shows the amplitude and frequency of an up-chirp.

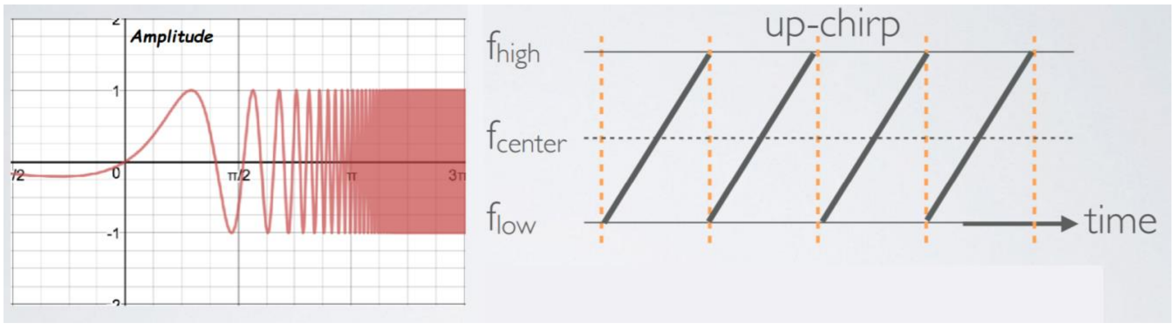


Figure 2.5: Up-chirp graphical representation [25].

A signal that contains a set of chirps is used as a carrier to encode digital information. The chirps are shifted periodically, and it is the role of frequency jumps to form a certain pattern that determines how the data is encoded on top of the chirps. Figure 2.6 shows how information is encoded using chirp signal modulation [25]. By a *symbol*, we mean a number of bits that can encode certain digital information. A symbol can hold one or more bits of data depending on the encoded information. If a symbol contains seven bits, we say that it can encode seven bits of raw data (from 0 to 127 in decimal). The number of bits is known also as the *spreading factor* and we write: a symbol can have 2^{SF} values. Given a piece of information in the form of a numerical value, the chirp signal is divided into 2^{SF} *chips* in order to encode this information into a modulated signal [25]. For example, the symbol is 10111111 (decimal value = 191), the number of raw bits that can be encoded by this symbol is 8 (SF=8), and the chirp signal is divided into up to $2^{SF} = 2^8 = 255$ chips as shown in Figure 2.7.

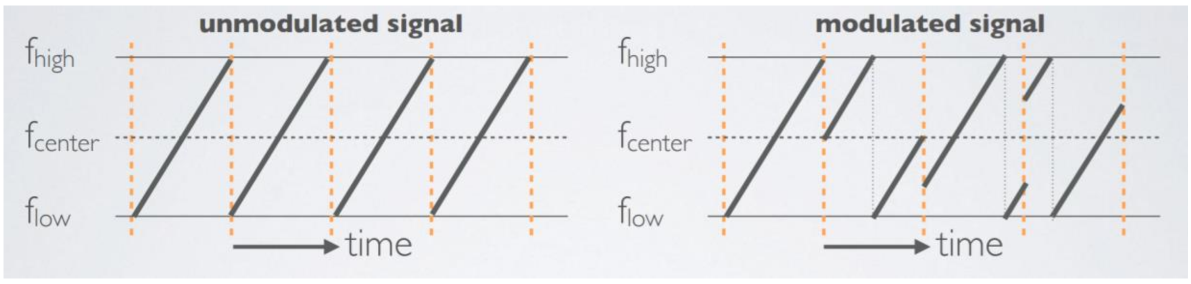


Figure 2.6: Message encoded on a chirp signal [25].

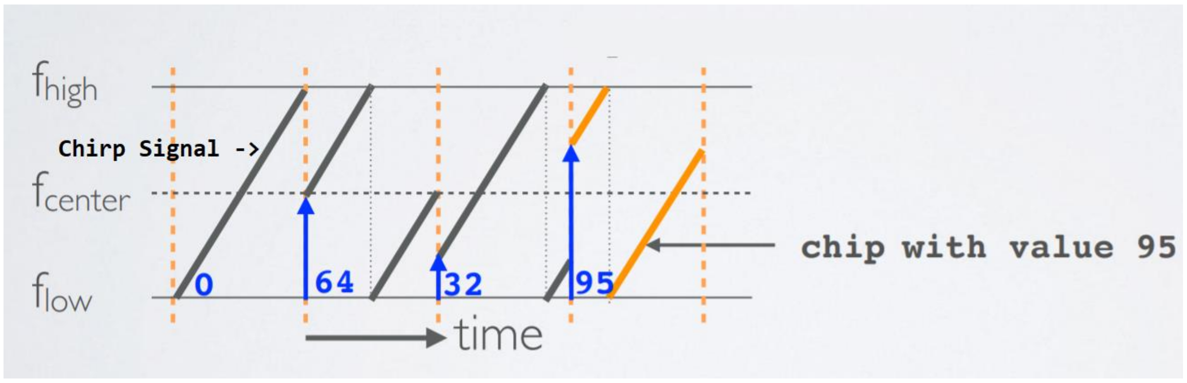


Figure 2.7: Chips representation [25].

Forward Error Correction (FEC) is the process where error correction bits are added to the transmitted data. These redundant bits help to restore the data when it gets corrupted by interference. The more error correction bits added, the easier the data can be corrected. However, by adding more error correction bits, more data are transmitted which decreases the battery life [25].

2.2.4. LoRa parameters

A LoRa device can be configured to use different Transmission Power (TP), Carrier Frequency (CF), Spreading Factor (SF), Bandwidth (BW), and Coding Rate (CR) to tune link performance and energy consumption [26].

1) *Transmission power*

TP on a LoRa radio can be adjusted from −4 dBm to 20 dBm, in 1 dB steps, but because of hardware implementation limits, the range is often limited to 2 dBm to 20 dBm. In addition, because of hardware limitations, power levels higher than 17 dBm can only be used on a 1% duty cycle [26].

2) *Carrier Frequency*

LoRa operates in the unlicensed ISM (Industrial, Scientific, and Medical) radio band that is available worldwide. CF is the center frequency that can be programmed in steps of 61 Hz between 137 MHz to 1020 MHz. Every country in the world has a specific frequency band that is allowed for Lora usage, for example, Europe uses 863 MHz to 870 MHz frequency range [26].

3) Spreading Factor

SF is the ratio between the symbol rate and chip rate. A higher SF increases the Signal to Noise Ratio (SNR), and thus sensitivity and range, but also increases the airtime of the packet. Each increase in SF divides the transmission rate by half and, hence, doubles transmission duration and ultimately energy consumption. SF can be selected from 6 to 12 [26].

4) Bandwidth

BW is the width of frequencies in the transmission band. Higher BW gives a higher data rate, but a lower sensitivity because of the integration of additional noise. A lower BW gives a higher sensitivity, but a lower data rate. Data is sent out at a chip rate equal to the bandwidth; a bandwidth of 125 kHz corresponds to a chip rate of 125 Kchips/sec. Although the bandwidth can be selected in a range of 7.8 kHz to 500 kHz, a typical LoRa network operates at 500 kHz, 250 kHz, or 125 kHz [26].

5) Coding rate

CR is the FEC rate used by the LoRa modem that offers protection against bursts of interference, and can be set to 4/5, 4/6, 4/7 or 4/8. A higher CR offers more protection, but increases time on air. Radios with different CR (and same CF, SF and BW), can still communicate with each other if they use an explicit header, as the CR of the payload is stored in the header of the packet, which is always encoded at CR 4/8 [26].

The unit of bandwidth is Hertz (Hz) which is the number of vibrations or wave cycles per second. In CSS, this term bandwidth is used interchangeably with the *chip rate* R_c [25]:

$$\mathrm{BW} = R_c = \text{chip rate (chips/sec)}. \tag{2.1}$$

For example: BW=125 kHz, gives Rc (chip rate) = 125 Kchips/sec.

The *Symbol Rate* (R_s) is calculated as follow [25]:

$$R_s = \mathrm{BW} \,/\, 2^{SF} = R_c \,/\, 2^{SF} \text{ (symbols/sec)}. \tag{2.2}$$

For example: BW=125 kHz, SF=7, gives $R_s = 125000 / 2^7 = 977$ symbols/sec.

To calculate the *data rate* (DR) also known as the *bit rate* (R_b) [25]:

$$R_b = SF * (\mathrm{BW}/2^{SF}) * 4/\,(4+CR) \text{ (bits/sec)}. \tag{2.3}$$

Where: *Code Rate* (CR) takes values from 1 to 4.

For example: SF=7, CR=1 BW=125 kHz, $R_b = 7 \times (125000 / 2^7) \times (4 / (4 + 1)) = 5.5$ Kbits/s.

2.3. Embedded system development

An embedded system can be broadly defined as a device that contains tightly coupled hardware and software components to perform a dedicated function with minimal human interaction, as a part of a larger system, and it is not intended to be independently programmable by the user. Most embedded systems interact directly with processes or the environment, making decisions based on their inputs. This makes it necessary that the system must be reactive, responding in real-time to process inputs to ensure proper operation. Besides, these systems operate in constrained environments where memory, computing power, and power supply are limited [27].

Another way to identify embedded systems is the use of cross compilers. While a cross compiler runs on a desktop or laptop computer, it creates code that does not. The cross-compiled image runs on the target embedded system. Since the code needs to run on a processor, the vendor for the target system usually sells a cross compiler or provides a list of available cross compilers to choose from. Many larger processors use the cross compilers from the GNU family of tools [28].

Embedded software compilers often support only C, or C/ C++. In addition, many embedded C++ compilers implement only a subset of the language (commonly, multiple inheritance, exceptions, and templates are missing). Regardless of the language used in the software design, object-oriented techniques can be practiced. The design principles of encapsulation, modularity, and data abstraction can be applied to any application in nearly any language. The goal is to make the design robust, maintainable, and flexible [28].

2.3.1. Embedded system components

When viewed from a general perspective, the hardware components of an embedded system include all the electronics necessary for the system to perform the function it was designed for. Therefore, the specific structure of a particular system could substantially differ from another, based on the application itself. Despite these dis-similarities, three core hardware components are essential in an embedded system: The Central Processing Unit (CPU), the system memory, and a set of input-output ports. The CPU executes software instructions to process the system inputs and to make the decisions that guide the system operation. Memory stores programs and data necessary for system operation. Most systems differentiate between program and data memories. The program memory stores the software programs executed by the CPU. Data memory stores the data processed by the system. The I/O ports allow conveying signals between the CPU and the external world [27].

Beyond this point, a number of other supporting and I/O devices needed for system functionality might be present, depending on the application. These include communication ports for serial and/or parallel data transfer, a user interface to interact with the user, sensors and electromechanical actuators to interact with the environment, data converters (Analog-to-digital (ADC) and/or Digital-to-Analog (DAC)) to allow interaction with analog sensors and actuators, system support components to provide essential services that allow the system to operate such as power supply and management components and clock frequency generators, other subsystems to enable functionality, this might include Application-Specific Integrated Circuits (ASIC), Field Programmable Gate Arrays (FPGA) and other dedicated units according to the complexity of the application. Figure 2.8 illustrates how these hardware components are integrated to provide the desired system functionality [27].

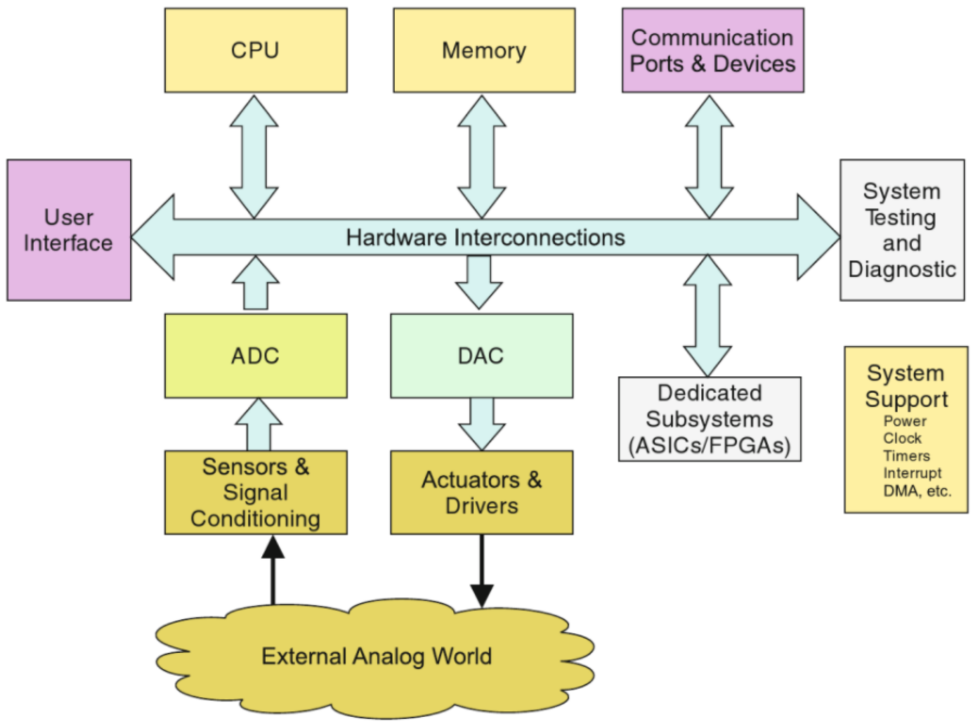

CPU
Memory
Communication Ports & Devices
User Interface
Hardware Interconnections
System Testing and Diagnostic
ADC
DAC
Dedicated Subsystems (ASICs/FPGAs)
System Support
Power
Clock
Timers
Interrupt
DMA, etc.
Sensors & Signal Conditioning
Actuators & Drivers
External Analog World

Figure 2.8: Hardware elements in embedded systems [27].

The software components of an embedded system include all the programs necessary to give functionality to the system hardware. These programs are referred to as the system *firmware* and are stored in some sort of non-volatile memory. Firmware is not meant to be modifiable by users, although some systems could provide means of performing upgrades. In high-performance embedded systems, system programs are organized around some form of operating system and application routines. The operating systems can be simple and informal in small applications, but as the application complexity grows, the operating system requires more structure and formality. In some cases, designs are developed around Real-Time Operating Systems (RTOS). In small embedded systems that do not require a high computing performance, the system is built on top of a single *microcontroller* chip that commands the whole application. These systems are roughly integrated, adding only a few hardware resources as needed, and operate with minimal or no maintenance. Software in such applications is typically single-tasked and does not require an RTOS [27].

2.3.2. Microcontroller unit

A microcontroller unit, abbreviated MCU, is developed using a CPU. The CPU is integrated along with memories of both types (program and data) and several types of peripherals, all embedded into a single integrated circuit or chip. This set of CPU, memory, and I/O within a single chip is what we call an MCU. The assortment of components embedded into an MCU allows for implementing complete applications requiring only a minimal number of external components or in many cases individually using only the MCU chip. Timer peripherals, input/output (IO) ports, interrupt handlers, and data converters are among those commonly found in most MCUs. The provision of such an assortment of resources inside the same chip is what has gained them the denomination of computers-on-chip. Figure 2.9 shows typical MCU architecture [27].

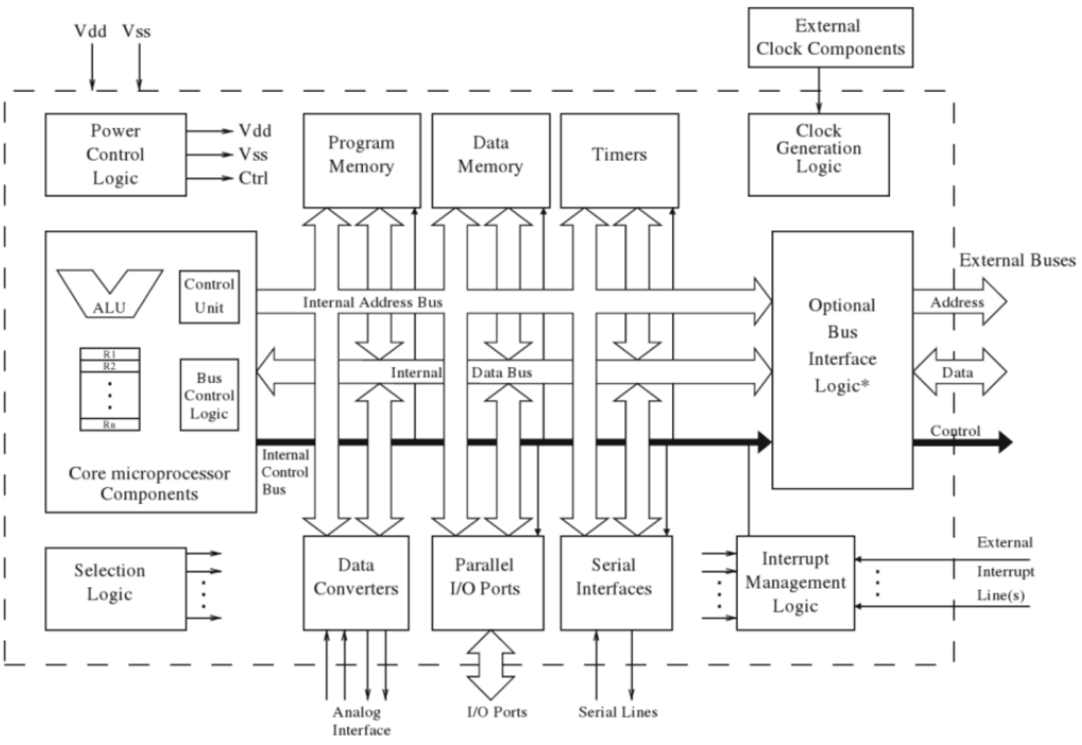


Figure 2.9: Microcontroller detailed architecture [27].

MCUs are usually marketed as family members. Each family is developed around a basic architecture that defines the common characteristics of all members. These include, among others, the data and program path widths, architectural style, register structure, base instruction set, and addressing modes. Features differentiating family members include the amount of on-chip data and program memory and the on-chip peripherals [27].

MCUs support Assembly programming language, however, most developers prefer C and/or C++ for the development of an MCU-based solution. C and C++ provide a wide range of advantages upon Assembly in terms of script readability, facility to write and debug the code, and the possibility to use high-level language methodologies like OOP. After compiling the firmware, a programmer is used to upload the code into the target MCU. A programmer is a piece of hardware that acts as a bridge between the user development environment like a laptop, and the target MCU, this allows the code to be placed in the appropriate memory location to be executed by the MCU to perform the intended operation.

2.3.3. Embedded solution design

The traditional design process of most embedded system solutions is developed around commercial off-the-shelf (COTS) parts. Although other methodologies exist, standard COTS methods are the default choice for embedded system designers having commercial components as the hardware resources to assemble an embedded solution. These methods allow for minimizing the hardware development costs through tight constraints on physical parts components, usually at the expense of larger software design and system verification cycles. Figure 2.10 shows a representative diagram of a typical COTS-based design process for an embedded solution. The diagram details the steps in the system conception, design, and prototyping stages. It also includes the estimated time duration of the different stages in the process. It can be observed that although hardware and software design could be carried in parallel, their functional verification can only happen on a functional prototype [27].

2.4. Data management

Data management is an administrative process that includes acquiring, validating, storing, protecting, and processing required data to ensure the accessibility, reliability, and timeliness of the data for its users. Organizations and enterprises are making use of Big Data more than ever before to inform business decisions and gain deep insights into customer behavior, trends, and opportunities for creating extraordinary customer experiences.

The best way to manage data, and eventually get the insights needed to make data-driven decisions, is to begin with a business question and acquire the data that is needed to answer that question. Companies must collect vast amounts of information from various sources and then utilize best practices while going through the process of storing and managing the data, cleaning and mining the data, and then analyzing and visualizing the data in order to inform their business decisions [29].

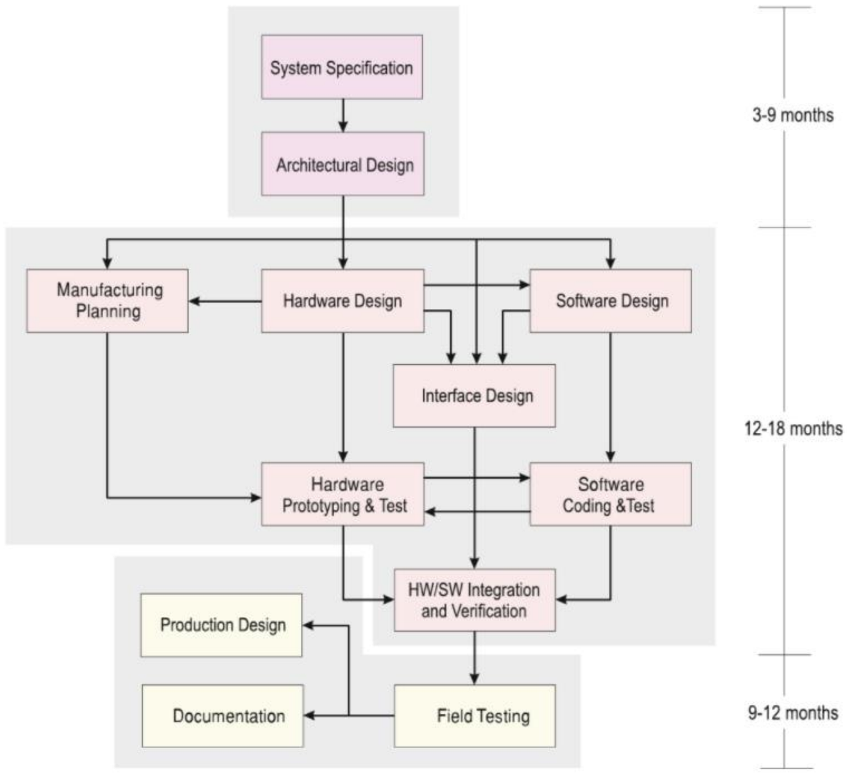


Figure 2.10: Embedded solution design work flow based on COTS parts [27].

A *database* is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a *database management system* (DBMS). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just database. Data within the most common types of databases in operation today is typically modeled in rows and columns in a series of tables to make processing and data querying efficient. The data can then be easily accessed, managed, modified, updated, controlled, and organized. Most databases use *structured query language (SQL)* for writing and querying data [30].

2.4.1. Time series database

A time series database (TSDB) is a database optimized for time-stamped or time series data. Time series data are simply measurements or events that are tracked, monitored, down sampled, and aggregated over time. This could be server metrics, application performance monitoring, network data, sensor data, events, clicks, trades in a market, and many other types of analytics data. A time series database is built specifically for handling metrics and events or measurements that are time-stamped. A TSDB is optimized for measuring change over time.

Properties that make time series data very different than other data workloads are data lifecycle management, summarization, and large range scans of many records [31]. Time series databases are the fastest-growing segment of the database industry over the past two years; this is illustrated in Figure 2.11. Time series databases are best suited for IoT applications where data is acquired over time. TSDB makes it easier to manage the gathered sensor's data in a clean and reliable way to make the most benefit of it.

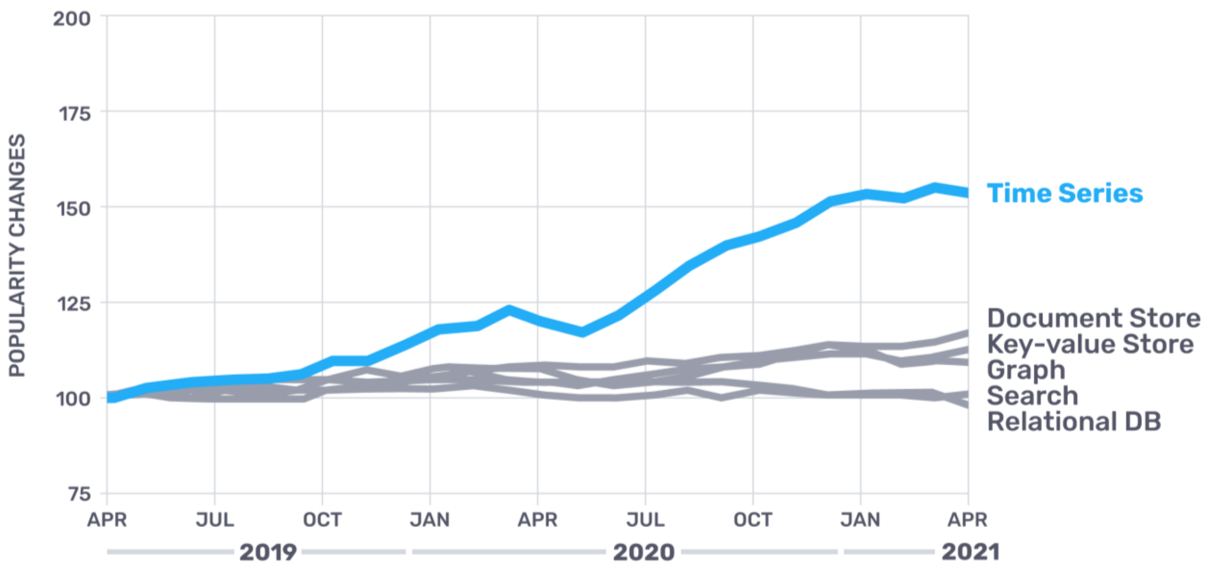


Figure 2.11: Database management systems popularity over the last two years [31].

Chapter 3: Design and Implementation

In this chapter, we will demonstrate detailed steps regarding the design process of the asset tracking solution. The first part covers the hardware aspect of the design, whereas the second part deals with the software design. Both the tracker node i.e., the transmitter, and the gateway i.e., the receiver, along with data storing and visualization are designed based on the information and techniques provided in previous chapters. For the prototype implementation, we used COTS modules for simplicity and availability, see section 2.3.3. However, the firmware was written from scratch using the C programming language. At the end of each part, a complete prototype implementation is built to demonstrate and discuss the obtained results of our solution.

3.1. Hardware design of the gateway and tracker node

The hardware design deals with all electronic components of both the tracker and the gateway. The goal is to build an embedded system that can send real-time GPS locations wirelessly over a specific period of time; on the other hand, a similar embedded system must be continuously listening to any data sent by nodes, then forward these data to a web application. MCUs are a good choice when it comes to designing embedded systems since they offer an excellent set of built-in peripherals that provide a wide range of options for interfacing purposes, as well as effective computation capabilities to run different tasks.

Our design was carried out based on the idea of using separate hardware blocks, also known as hardware modules, each for a specific purpose, then to connect all parts together to perform the desired operation on both the tracker and the gateway sides. Generally, components like GPS receivers and LoRa transceivers come as a standalone System-on-Chip (SoC), such systems contain an integrated computing unit that internally performs the required task and provides a mean of communication to developers for configuration and interfacing. Communication protocols such as *Universal Asynchronous Receiver and Transmitter (UART), Serial Peripheral Interface (SPI),* and *Inter-Integrated Circuit* (*I^2C)* are widely used for this purpose.

We chose to work with separate hardware modules for the reason that such a technique reduces the complexity of the hardware design as each module is delivered as a compact chip with only the required pins for interfacing. On the other hand, the major advantage of using hardware modules is to reduce the computation complexity off the MCU, so instead of having a single MCU responsible to perform all computations, each module runs the specific task which is designed to deliver. For example, in our design, the localization operation was assigned to the GPS module which internally contains large circuits with many filters and performs complex computations just to provide GPS locations as a ready-to-use raw data through its communication protocol.

Starting with the tracker side, the system includes a GPS receiver for localization and a LoRa transceiver to send the acquired location wirelessly to the gateway. An MCU is used to interface both the GPS and LoRa modules. The MCU receives GPS data from the GPS module, extracts the location data, places it into an appropriate format then transfers it to the LoRa module in which the data is converted into an RF packet and sent over the air to the gateway. Figure 3.1 shows the block diagram of the tracker node functionality.

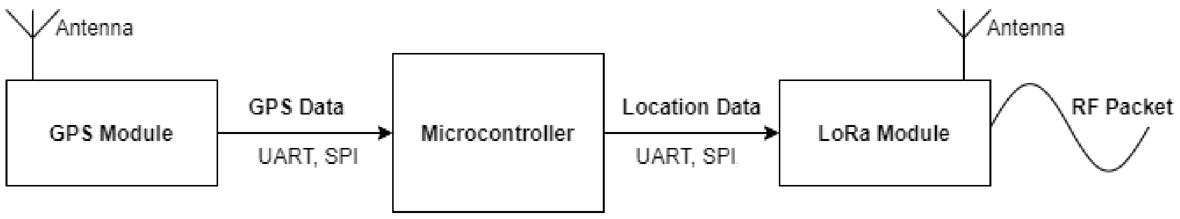


Figure 3.1: Block diagram of the tracker node functionality.

On the other hand, the gateway consists of a LoRa module to receive RF packets holding the location data and a network interface module to transfer this data to the web application; Wi-Fi or Ethernet modules can be used as network interface components. Similarly, Wi-Fi and Ethernet modules are provided as SoCs that are ready to use by the developer and allow interfacing using communication protocols mentioned before. An MCU is used to interface the LoRa module with the network interface module, hence, receiving location data from the tracker, convert it into a network-like packet format then transfers it to the network interface module where it is forwarded to the web application. Figure 3.2 shows the block diagram of the gateway functionality.

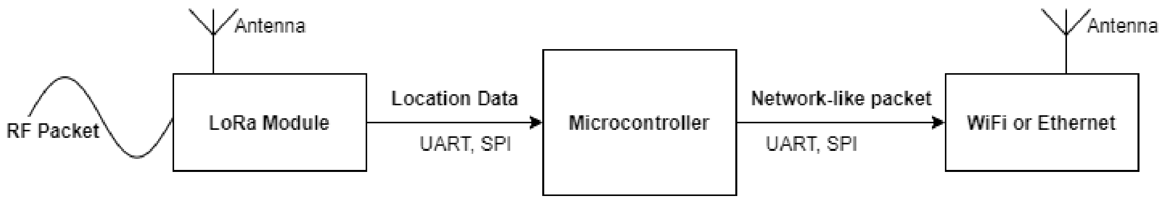


Figure 3.2: Block diagram of the gateway functionality.

In our hardware design process, we have decided to use both SPI and UART communication protocols on each side interchangeably so that the gateway and the tracker embedded systems can perform their intended functions properly without any data interference, meaning that each communication port is responsible for either data transmission or reception. For the UART hardware design, we have connected the transmitter (TX) and receiver (RX) lines of the corresponding GPS and Wi-Fi/Ethernet modules to the RX and TX lines of the MCU unit respectively. Furthermore, the master in – slave out (MISO), the master out – slave in (MOSI), the serial clock (SCK), and slave select (SS) lines of the SPI port of the relevant LoRa module were connected to their opposite lines on the MCU side. Figure 3.3 illustrates the hardware designs of both the gateway and the tracker node with all required connections.

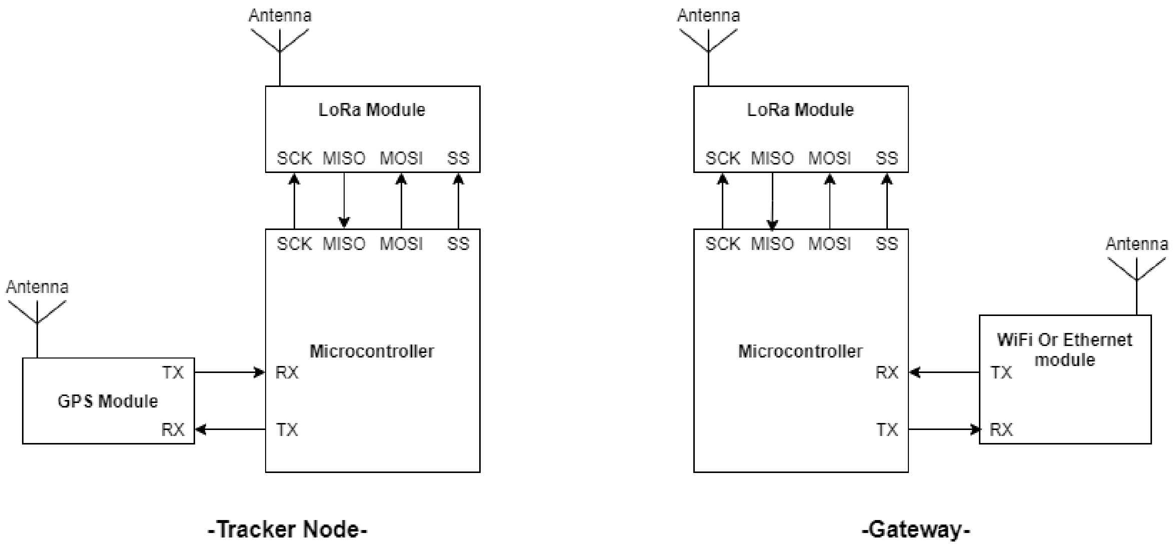


Figure 3.3: Tracker node and Gateway hardware designs.

3.2. Hardware implementation of the gateway and tracker node

In an effort to test the validity of our hardware design, we were supposed to build two embedded system prototypes for the gateway and the tracker node. The prototypes used hardware modules as building blocks which were selected based on our hardware design approach from one side, and the availability of hardware components in the market from the other side. The following sections introduce the hardware modules and computing units we used in our hardware implementation.

3.2.1. Atmel ATmega328P

The Atmel ATmega328P is a low-power 8-bit MCU based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega328P achieves throughputs approaching 1MIPS per MHz allowing the system designer to optimize power consumption versus processing speed. The Atmel ATmega328P provides the following features: 32K bytes of in-system programmable flash with read-while-write capabilities, 1K bytes EEPROM, 2K bytes SRAM, 23 general-purpose I/O lines, 32 general purpose working registers, three flexible Timer/Counters with compare modes, internal and external interrupts, a serial programmable USART, an SPI serial port, and a 6-channel 10-bit ADC. The ATmega328P reaches 16 MHz clock speed on a voltage range from 2.7V to 5.5V and can operate on temperature range between –40°C and +125°C, which makes it good choice for automotive applications [34]. Figure 3.4 shows the ATmega328P pinout and SMD chip.

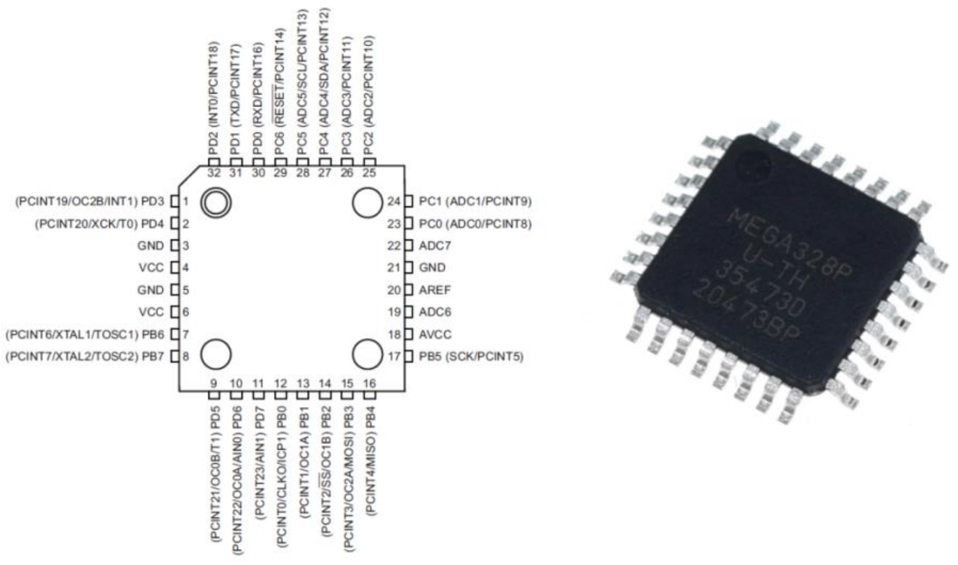


Figure 3.4: ATmega328P pinout and SMD chip [34].

The device is manufactured using Atmel high density non-volatile memory technology. The on-chip ISP flash allows the program memory to be reprogrammed in-system through an SPI serial interface, by a conventional non-volatile memory programmer, or by an on-chip boot program running on the AVR core. The boot program can use any interface to download the application program in the application flash memory. The ATmega328P AVR is supported with a full suite of program and system development tools including C compilers, macro assemblers, program debugger/simulators, in-circuit emulators, and evaluation kits [34].

We opted to use the ATmega328P MCU because it allows access to a huge number of documentations and resources besides the large developers' community using this MCU family; this gives the advantage to implement different prototypes with the help of the available examples and the experiences of the Atmel developers' community on both the hardware and software aspects. Beside the fact that the ATmega328P is used for automotive applications which directly meets our asset tracking use case, it integrates mainly all communication protocols that were chosen in our hardware design; these are the SPI and the UART.

1) *Serial Peripheral Interface*

The SPI allows high-speed synchronous data transfer between the ATmega328P and peripheral devices or between several AVR devices, the data communication occurs between what is called *Master* and *Slave* peripherals. The interconnection between master and slave CPUs with SPI is shown in Figure 3.5. The system consists of two shift registers and a master clock generator. The SPI master initiates the communication cycle when pulling low the slave select (SS) pin of the desired Slave. Master and Slave prepare the data to be sent in their respective shift registers, and the master generates the required clock pulses on the SCK line to interchange data. Data is always shifted from master to slave on the master out – slave in, MOSI, line, and from slave to master on the master in – slave out, MISO, line. After each data packet, the master will synchronize the slave by pulling high the slave select, SS, line [34].

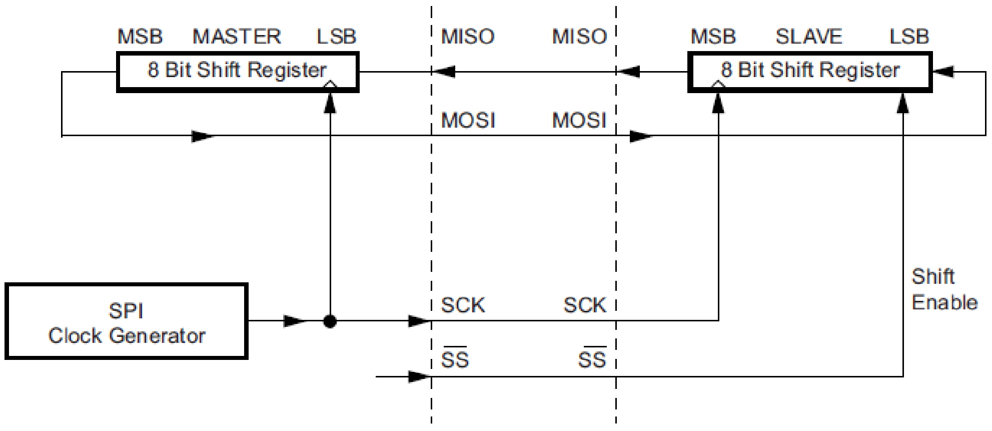


Figure 3.5: SPI Master-slave interconnection between MCU and peripheral [34].

2) *Universal Asynchronous Receiver and Transmitter*

The UART is a highly flexible serial communication protocol. It consists of three main parts: a baud rate generator for setting data sampling speed, a transmitter and a receiver to exchange data from and into external peripherals. For the ATmega328P, the UART features a full-duplex operation meaning that sending and receiving data can be performed simultaneously since both receiver and transmitter include independent registers. The UART has to be initialized before any communication can take place. The initialization process consists of setting the baud rate, setting frame format, and enabling the transmitter or the receiver depending on the usage. The interconnection between two devices using UART is shown in Figure 3.6.

Data transmission is initiated by loading the transmit buffer with the data to be transmitted. The buffered data in the transmit buffer will be moved to the shift register when it is ready to send a new frame. The shift register is loaded with new data if it is in the idle state (no ongoing transmission) or immediately after the last stop bit of the previous frame is transmitted. When the shift register is loaded with new data, it will transfer one complete frame at the rate given by the baud register. On the other hand, the receiver starts data reception when it detects a valid start bit. Each bit that follows the start bit will be sampled at the baud rate, and shifted into the receive shift register until the first stop bit of a frame is received. A second stop bit will be ignored by the receiver. When the first stop bit is received, i.e., a complete serial frame is present in the receive shift register, the contents of the shift register will be moved into the receive buffer [34].

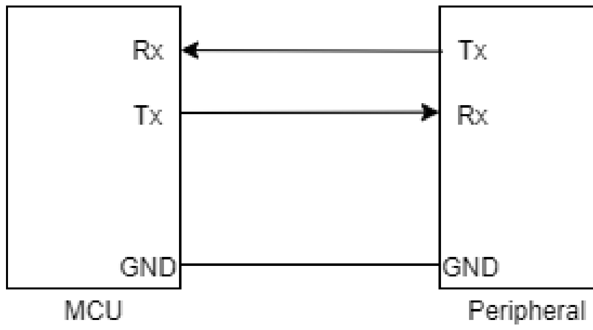


Figure 3.6: UART circuit interconnection between MCU and peripheral.

For the implementation process, we have used the Arduino UNO board, which is an ATmega328P-based development board designed for fast prototyping. We have selected the Arduino board first because it is widely available in the market at a cheap price, it includes also a built-in USB-to-Serial chip that allows uploading codes to the MCU without the need for any external hardware (programmer), in addition to some components such as voltage regulators, a reset button, power outputs, and LED indicators for more sophisticated use. The Arduino UNO was used to implement the tracker node part, which is to interface the GPS module and the LoRa transceiver to build an IoT node that transmits asset locations over the air.

3.2.2. Espressif ESP8266

The ESP8266 is a low-cost Wi-Fi microchip, with a full TCP/IP stack and MCU capability, produced by Espressif Systems. The ESP8266 delivers a highly integrated Wi-Fi SoC solution to meet users' continuous demands for efficient power usage, compact design, and reliable performance in IoT industry. With the complete and self-contained Wi-Fi networking capabilities, ESP8266 can act either as a standalone application or as the slave to a host MCU. When ESP8266 hosts the application, it promptly boots up from the flash. The integrated high-speed cache helps to increase the system performance and optimize the system memory. Also, ESP8266 can be integrated into any MCU design as a Wi-Fi adaptor through SPI or UART interfaces. Besides the Wi-Fi functionalities, the ESP8266 integrates an enhanced version of Tensilica's L106 Diamond series 32-bit processor which achieves extra-low power consumption and reaches a maximum clock speed of 160 MHz. It can be interfaced with external sensors and other devices through the GPIOs [35]. Figure 3.7 shows the ESP8266 module pinout and SMD chip.

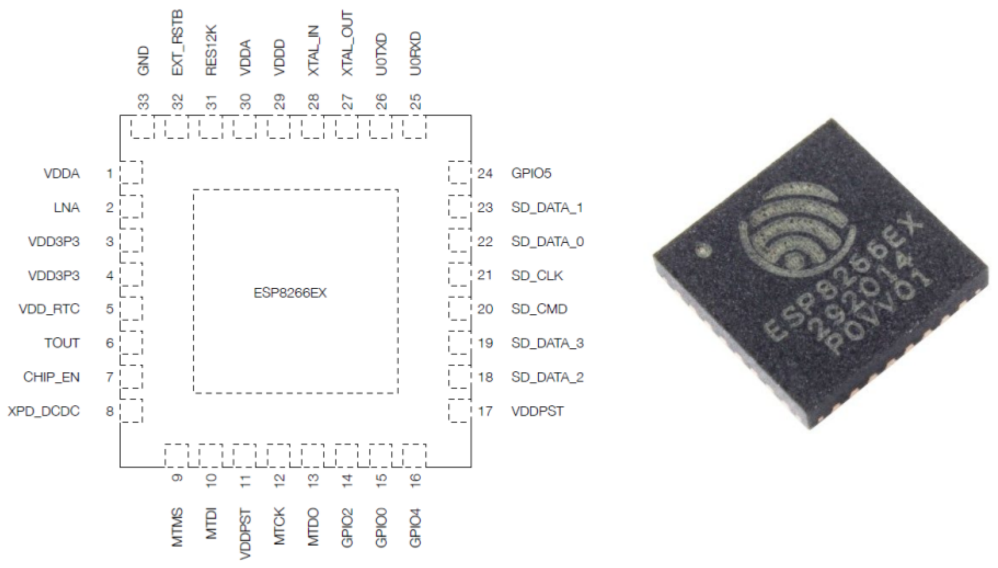


Figure 3.7: ESP8266 chip pinout and SMD chip [35].

The ESP8266 Software Development Kit (SDK) is an IoT application development platform created by Espressif for developers and includes two types: Non-OS SDK and RTOS SDK. ESP8266 implements TCP/IP and full 802.11 b/g/n WLAN MAC protocol. In the world of wireless communication, the term Wi-Fi is related to wireless access in general, despite the fact that it is a specific trademark owned by the Wi-Fi Alliance, a group dedicated to certifying that Wi-Fi products meet the IEEE's set of 802.11 standards. IEEE 802.11 is a set of technical guidelines for implementing Wi-Fi communication protocol [36]. On the other hand, TCP/IP is a widely used Internet protocol suite, which is the conceptual model and set of communication protocols used in the Internet and similar computer networks. TCP/IP is a set of standard Internet communication protocols that allow digital computers to communicate over long distances. The Internet is a packet-switched network, in which information is broken down into small packets, sent individually over many different routes at the same time, and then reassembled at the receiving end. TCP is the component that collects and reassembles the packets of data, while IP is responsible for making sure the packets are sent to the right destination [37].

For our implementation, we have used the NodeMCU board which is a ready-to-use development board based on the ESP8266 module designed for fast prototyping of IoT projects. Similar to Arduino, The NodeMCU does not require any external programming hardware to upload codes to the MCU. Furthermore, NodeMCU defines an open-source firmware that is built on top of the Espressif Non-OS SDK; this firmware makes it easier to develop and upload codes in a flexible and reliable way especially during the prototyping process. NodeMCU was used to implement the gateway part of our solution, which is to receive RF data through the LoRa transceiver then transfer these data to the web application using WiF for further use.

It was more practical to choose an Ethernet module since the gateway side needs a stable and constant network connection which may be achieved by Ethernet rather than Wi-Fi. However, we have decided to go with the ESP8266 over other hardware modules as it comprises a network connection interface with a programmable MCU in a single chip, this would greatly reduce the hardware connections and allow more flexibility for interfacing. Since the ESP8266 combines a Wi-Fi module with an integrated MCU, we would have the ability to choose between either SPI or UART to interface the LoRa transceiver, unlike in the hardware design where a UART port was only dedicated to connecting the MCU unit to the network interface module which would not be the case in the implementation.

3.2.3. NEO-6M U-blox GPS module

The NEO-6M module is a stand-alone GPS receiver featuring a high-performance positioning engine. This flexible and cost-effective receiver offers numerous connectivity options in a miniature 16 x 12.2 x 2.4 mm package. Its compact architecture and power and memory options make the NEO-6 module ideal for battery-operated mobile devices with very strict cost and space constraints [32]. Figure 3.8 shows the GPS module with its Antenna.

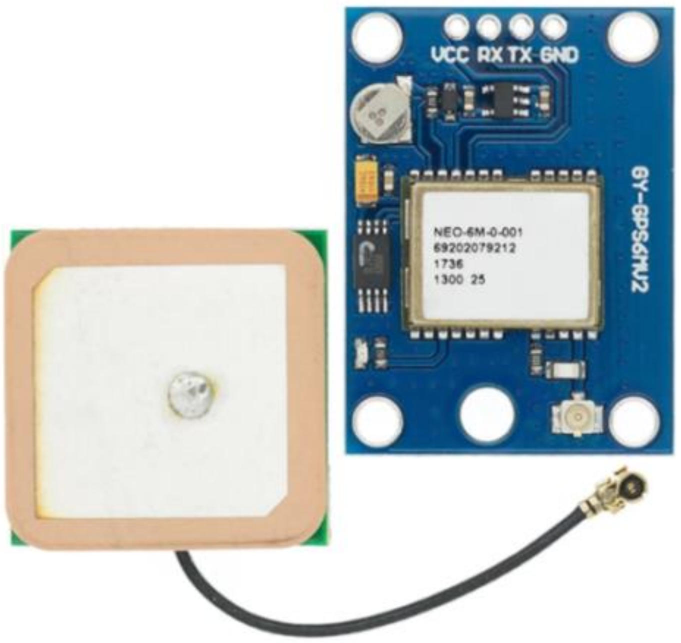


Figure 3.8: NEO-6M GPS module with antenna [32].

The NEO-6M operates on a maximum voltage of 3.6V and includes one configurable UART interface for serial communication. The UART port is used to transmit GPS measurements which are structured in the NMEA data format; this will be described in greater details later in the software implementation section. By default, the GPS module measures its location at a frequency of 1Hz and then transmits this information through the serial port at 9600 bits/sec baud rate. As a first step to test the functionality of our GPS module, we have powered up the NEO-6M by attaching a 3.6V power source to its power pins VCC and GND, this module contains a blue LED which indicates whether the receiver is working properly or not; after waiting for a while, the LED started blinking as an indication that our module hardware did not contain any internal hardware fault and was ready for interfacing with the MCU. The NEO-6M GPS receiver was integrated into the tracker node embedded system prototype for localization purposes.

3.2.4. SX1278 LoRa Transceiver

The SX1278 LoRa transceiver is a cheap LoRa module mainly used for long-range spread spectrum communication. It for provides ultra-long range spread spectrum communication and high interference immunity whilst minimizing current consumption. For maximum flexibility, the user may decide on the spread spectrum modulation bandwidth, spreading factor, and correction rate (CR). The SX1278 offers bandwidth options ranging from 7.8 kHz to 500 kHz with spreading factors ranging from 6 to 12 and covering carrier frequencies from 137.0 MHz to 525.0 MHz [33]. Figure 3.9 shows the LoRa transceiver pin configuration. The SX1278 is a half-duplex transceiver that can act either as a receiver or a transmitter; ideally, it operates at a voltage of 3.3V and uses SPI communication protocol to interface with external computation devices for reading or writing data. The module also contains a built-in cyclic redundancy check (CRC) algorithm that can be used to detect errors in the incoming payloads.

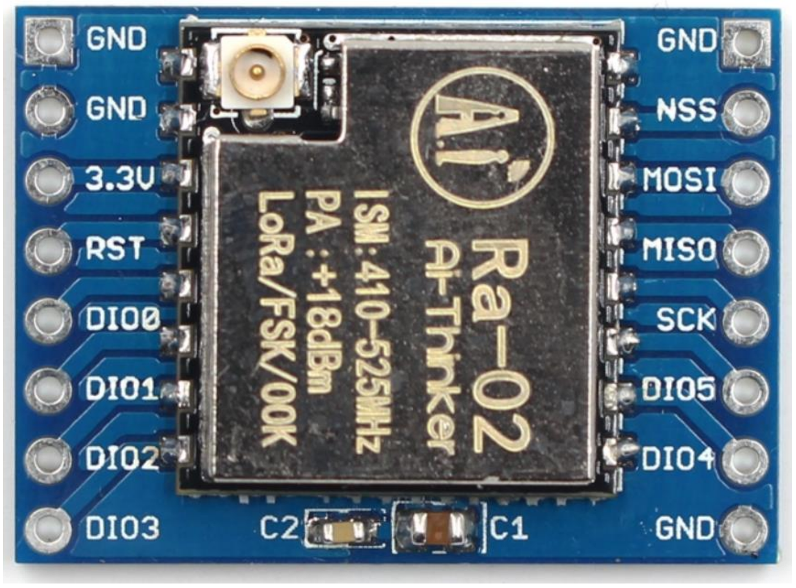


Figure 3.9: SX1278 LoRa Transceiver pin configuration [33].

A simplified outline of the transmitting and receiving processes is shown in Figure 3.10. Here we see that the LoRa modem has an independent dual-port First-in First-out (FIFO) data buffer that is accessed through the SPI interface. Configuration registers give access to modify different parameters such as carrier frequency and spreading factor; this allows flexible performance optimization depending on the application and usage.

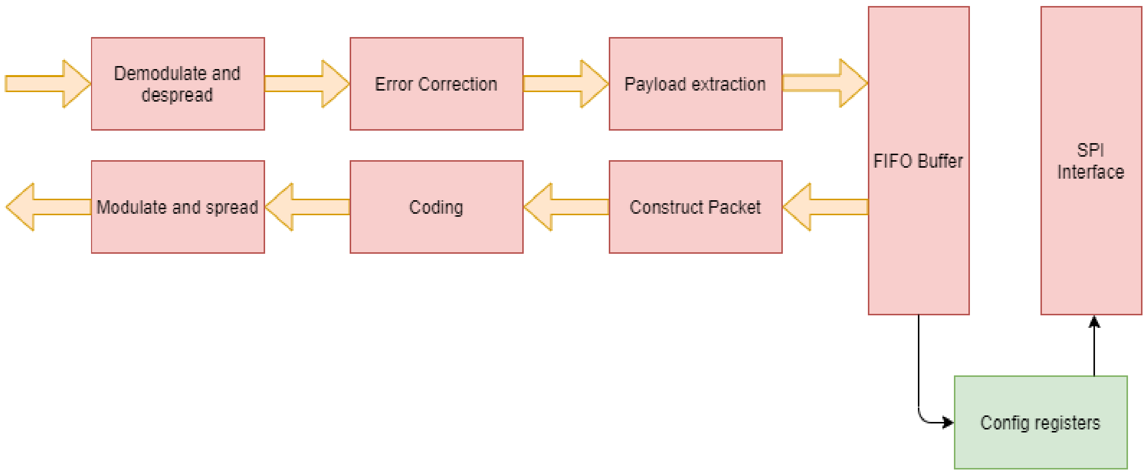


Figure 3.10: LoRa modem connectivity [33].

The SX1278 is equipped with a 256 byte RAM, also referred to as FIFO Data buffer. It is fully customizable by the user and allows to read the received data, or to write the transmitted data. All the access to the LoRa FIFO data buffer is done via the SPI interface. By default, the device is configured at power-up so that half of the available memory is occupied for reception (*RegFifoRxBaseAddr* initialized at address 0x00), and the other half is occupied for transmission (*RegFifoTxBaseAddr* initialized at address 0x80) as shown in Figure 3.11. However, due to the contiguous nature of the FIFO data buffer, the base addresses for transmission and reception operations are fully configurable across the 256 byte memory area. The FIFO data buffer location to be read from, or written to, via the SPI interface is defined by the address pointer *RegFifoAddrPtr*. Upon reading or writing to the FIFO data buffer, the address pointer will then increment automatically, therefore, it is necessary to initialize all these pointer registers to the corresponding addresses before any read or write operation [33].

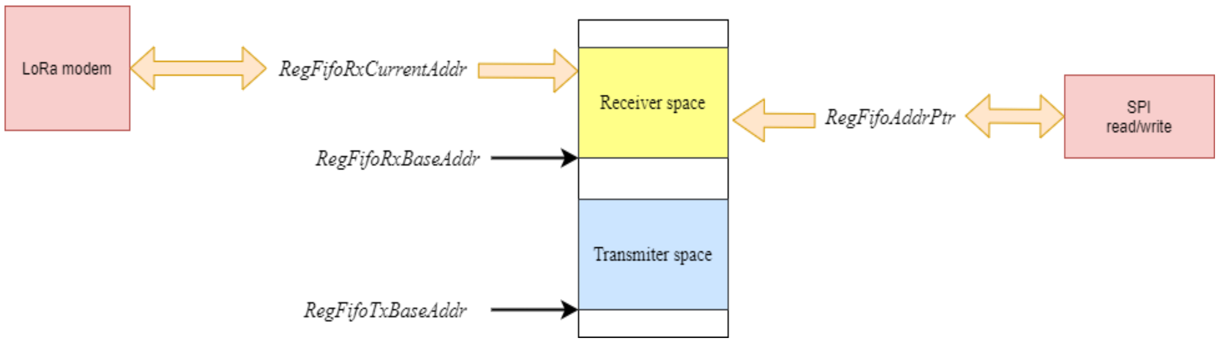


Figure 3.11: SX1278 internal memory mapping [33].

3.2.5. Tracker node prototype implementation

Based on the hardware modules that we described in sections 3.2.1, 3.2.3, and 3.2.4, we have built a tracker node that would allow collecting location data from mobile assets. The tracker prototype circuit consists of an ATmega328P MCU as a controlling and interfacing unit, the NEO-6M GPS module, and the SX1278 LoRa transceiver. The tracker node localizes the asset mounted on it using the GPS module, this data is transferred to the MCU using UART protocol and then transmitted to the gateway using the LoRa module which reads data from the ATmega328P via SPI; The LoRa module acts as a transmitter at this side of the implementation. First, we have connected the UART port pins TX and RX of the NEO-6M to the PD0 (RX) and PD1 (TX) pins of the ATmega328P respectively. For the SX1278, we have attached the SPI port pins NSS, MOSI, MISO, and SCK to pins PB2 (SS), PB3 (MOSI), PB4 (MISO), and PB5 (SCK) of the ATmega328P, see Figure 3.4. Since we have used an Arduino board for our prototype, we could power our modules directly from the 3.3V power output provided by the board. Figure 3.12 shows the tracker prototype circuit connections and Figure 3.13 shows the prototype implementation of the tracker.

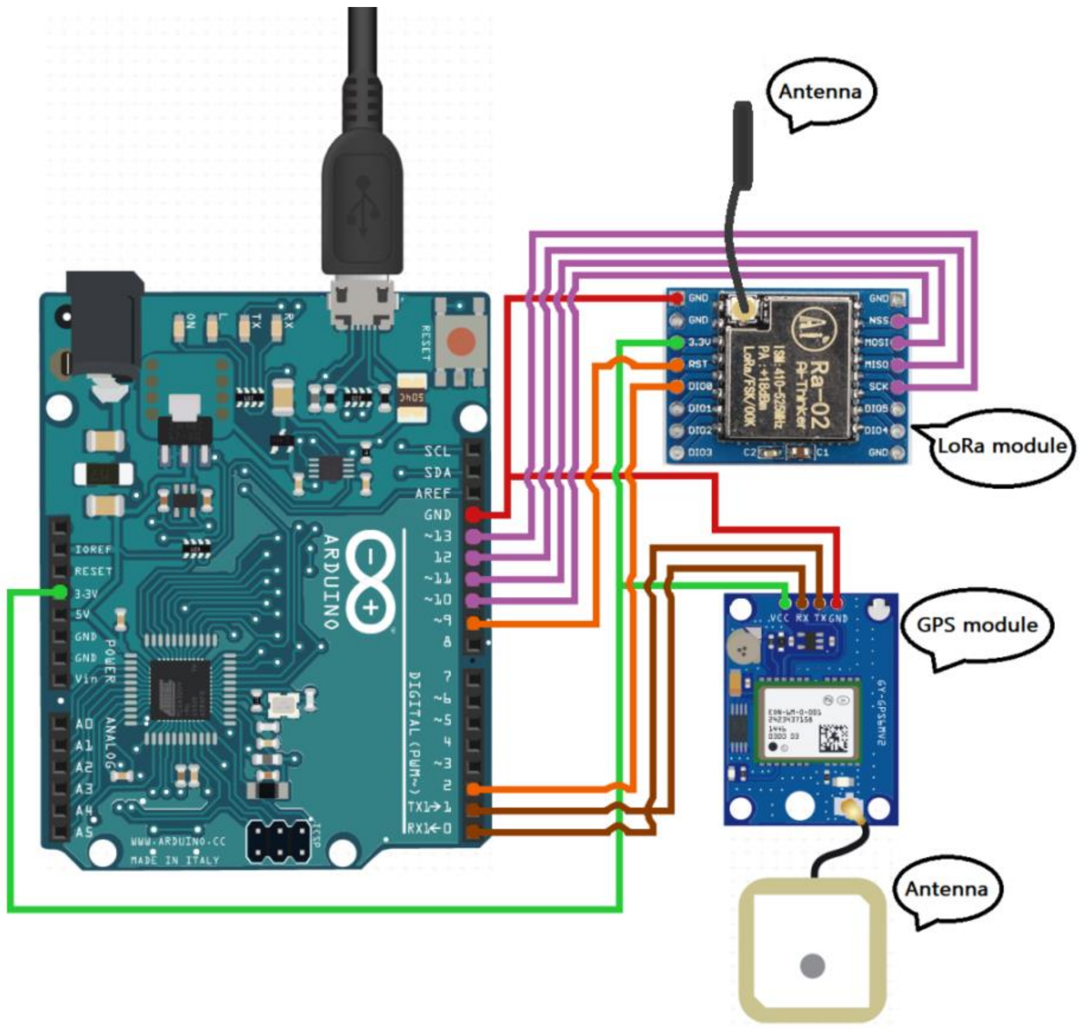


Figure 3.12: Tracker prototype circuit connections.

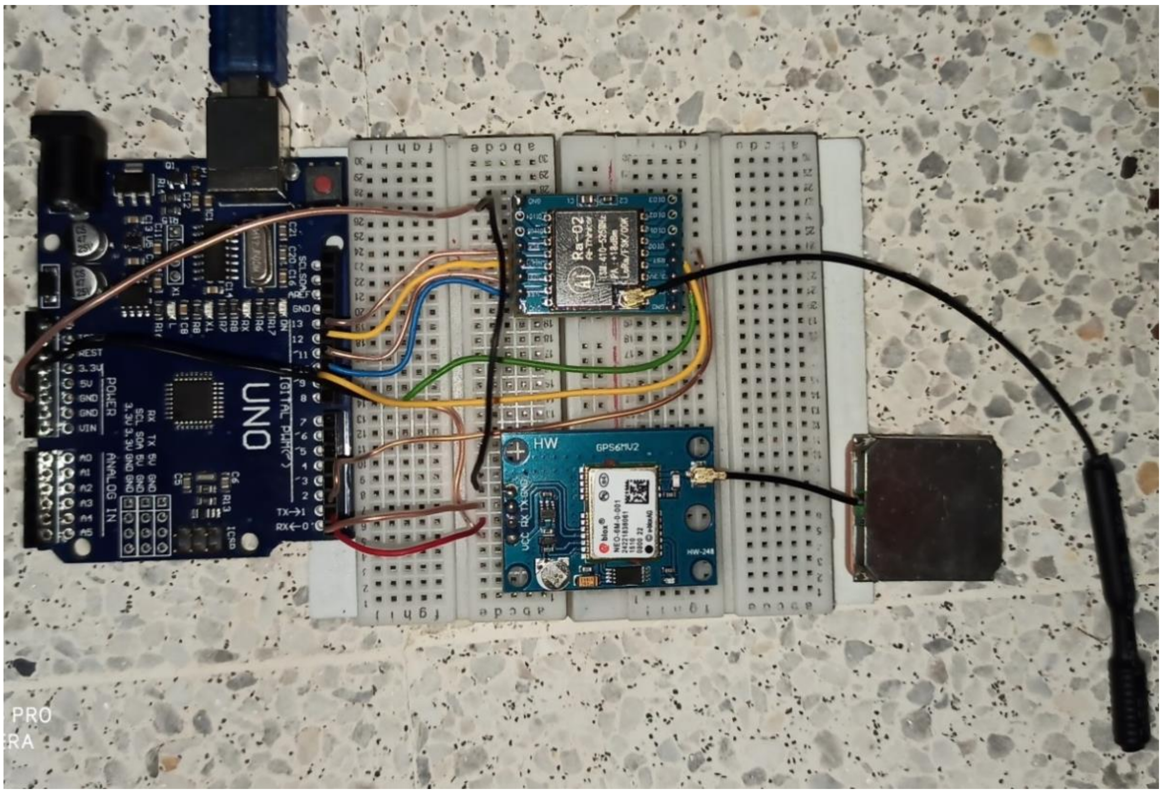


Figure 3.13: Tracker prototype implementation.

At this point of the implementation, the MCU did not contain any firmware inside. However, we could get an insight that the GPS module was working properly since it included an LED indicator, unlike the LoRa module which did not provide the option of checking its functionality at this stage, this would be done later in the software implementation part. A piece of software would be used to test the functionality of sending and receiving data using the SX1278 and the check location data from the NEO-6M as well.

3.2.6. Gateway prototype implementation

We have implemented the gateway circuit which consists of an ESP8266 module interfaced with the SX1278 LoRa transceiver. The LoRa transceiver receives RF packets sent wirelessly from nodes; these packets contain information about real-time locations of assets. The data is then transferred using SPI protocol to the ESP8266 in which it is converted into network-like packets then sent to the web application via Wi-Fi. We have connected the SPI port pins NSS, MOSI, MISO, and SCK of the SX1278 to pins MTDO (SS), MTCK (MOSI), MTDI (MISO), and MTMS (SCK) of the ESP8266 Wi-Fi module, see Figure 3.7. The NodeMCU includes a 3.3V power output pin which is enough to feed the LoRa module during the testing operation. After connecting all necessary pins, we could not have any idea regarding the functionality of the SX1278 as the gateway at this point does not run any firmware so; this would be covered later in the software implementation and testing part. As expected, the gateway prototype circuit contained only few connections compared to the tracker's circuit implementation.

Figure 3.14 shows the gateway prototype circuit connections and Figure 3.15 shows the prototype implementation the gateway.

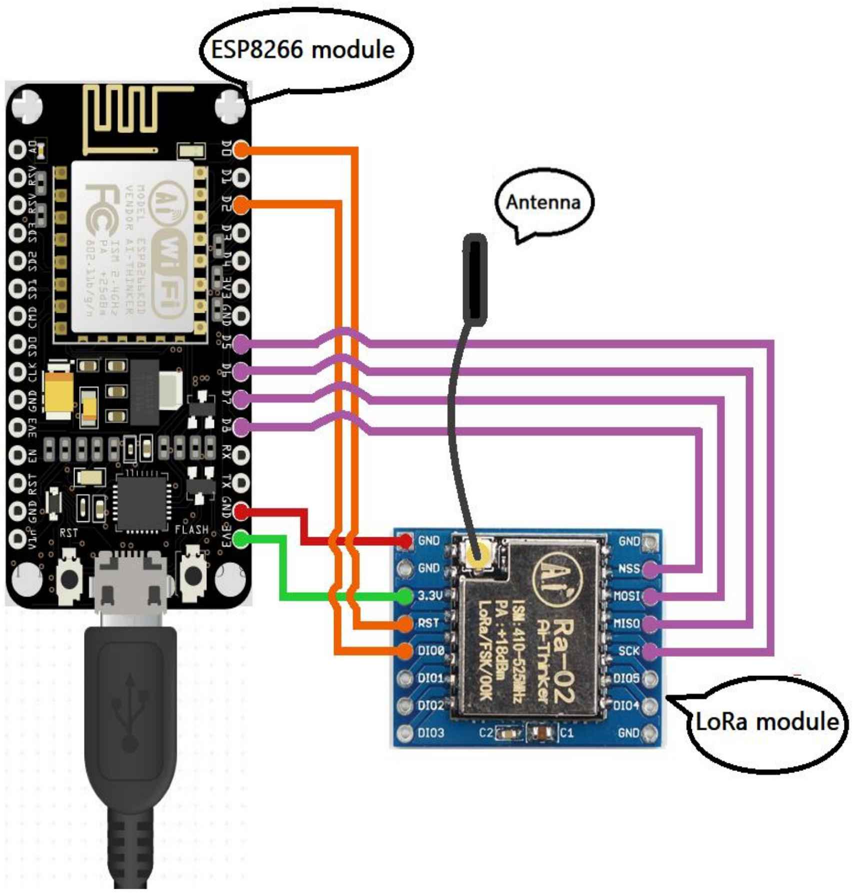


Figure 3.14: Gateway prototype circuit connections.

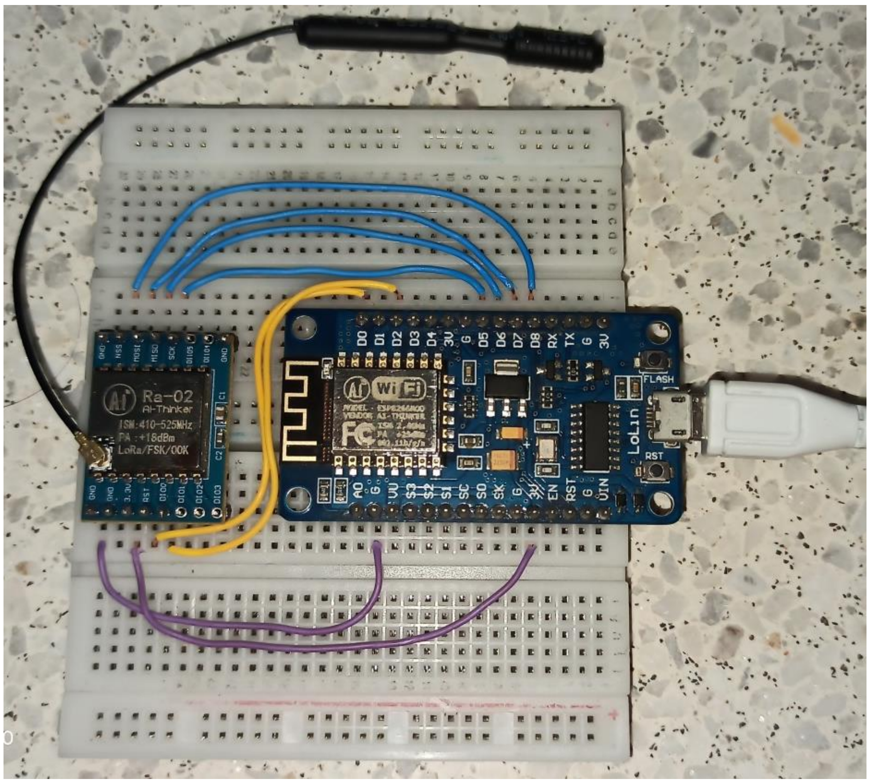


Figure 3.15: Gateway prototype implementation.

3.3. Software design of the gateway and tracker node

The software design deals with the algorithm and logic that run the whole system. In our case, it was necessary to build a piece of firmware that would be able to control different parts of our solution in order to realize the intended behavior from the gateway and the tracker node. Nevertheless, it is important to know that we will not be designing an algorithm to solve a mathematical problem, rather than designing software modules to drive the process of sending and receiving data in a reliable and effective way based on our hardware design. The tracker node and the gateway must operate coordinately to localize assets in real-time, thus solving the problem of asset tracking within industrial environments.

The core operation of the tracker node was to detect asset locations using the GPS receiver, put the received data in an appropriate format, and then send it to the gateway using LoRa transceiver, which acts as a transmitter on this side of the solution. Our software design for the tracker node started by setting up the needed communication protocols which would be used to interact with the LoRa and GPS modules, these are the SPI and UART protocols. The data transmission operation was based on the condition that the node must receive a valid location before moving to the transmission stage. If valid GPS data is received by the GPS module, the system has to extract only the location field from the whole GPS data then moves on to the transmission step using the LoRa module. Moreover, the asset tracking operation must be done over a specific period of time meaning that data is sent once every n seconds depending on the user’s specifications. The software design flowchart for the tracker node is presented in Figure 3.16.

The gateway function consists of receiving RF packets from tracking nodes using the LoRa module, which acts as a receiver on this side, converts these packets again into the appropriate format which would be compatible with the web application, then forwards the data to the web application using the network interface module. Likewise, the gateway has to set up communication protocols to allow communication between the LoRa receiver and the network interface module, as well as to initialize parameters for network communication to allow the interaction with the web application side. Our design implements an algorithm that continuously checks for any available data sent from nodes; if any data is received it is promptly forwarded to the web application using the network interface module. Figure 3.17 shows the software design flowchart on the gateway side.

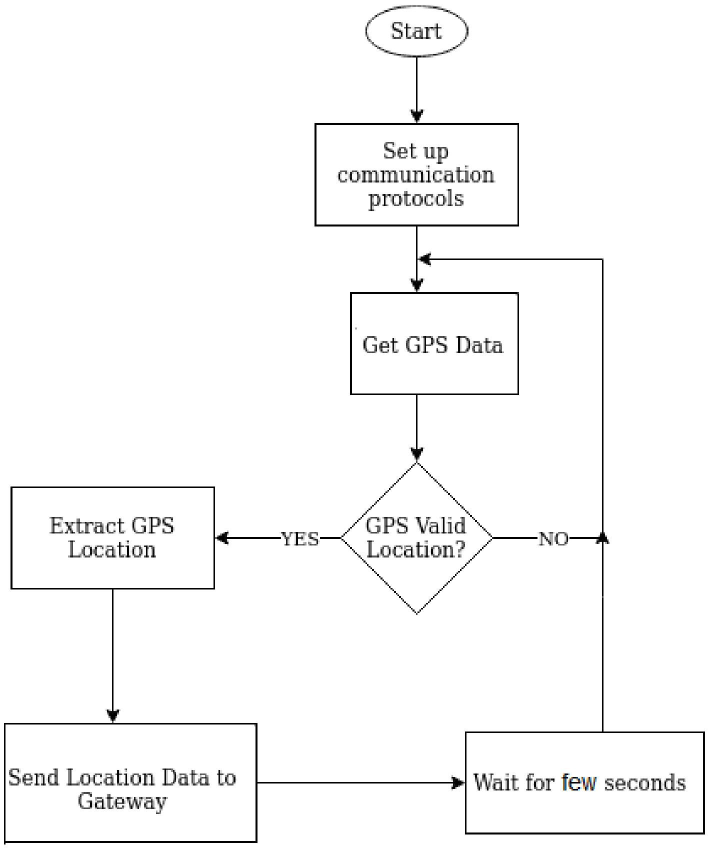


Figure 3.16: Tracker software design flowchart.

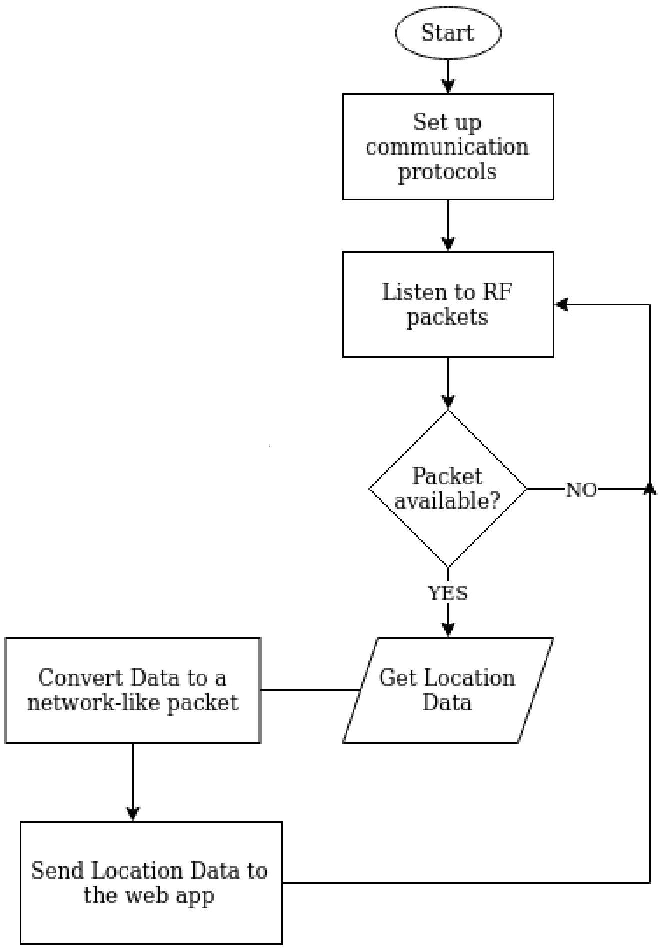

Start
Set up communication protocols
Listen to RF packets
Packet available?
NO
YES
Get Location Data
Convert Data to a network-like packet
Send Location Data to the web app

Figure 3.17: Gateway software design flowchart.

Since we were focusing only on the embedded system design of the solution, the web application will not be included in the design process; however, we will be using existing storing and visualizing tools that meet the requirements of our solution, these tools will be introduced later in the implementation section. The web application should be able to store location data over time as well as visualize it on a graphical map to show the exact location of an asset in real-time. It consists of a data storing application, hence a database system, which sources data to a visualization application to show asset locations on a graphical map in real-time. We have decided to work with two separate systems, one for data storage and one for data visualization, as most of the existing database management systems do not provide integrated visualization options, so we had to add an extra layer just for visualization; on the other hand, the visualization process was one of the core functions that must be included in the asset tracking solution, so it was more practical to choose a dedicated system just for visualization which would give a better user experience. We had to choose a database system that would be able to communicate with the gateway in order to receive data via the network from one side and to forward this data to the visualization application from the other side. Also, we have chosen to work with time-series databases because it would be the best choice to work with IoT data that change over time; a timer-series database would allow a better understanding of the captured data with more flexibility to store, process, and query data over time.

3.4. Software implementation of the gateway and tracker node

Our hardware implementation was based on MCUs as controlling units which allow a variety of choices when it comes to programming languages for developing the firmware. In our implementation of the tracker node, we have used the C programming language, which is one of the widely used languages to write firmware for MCUs. The C language as a mid-level programming language provides two critical characteristics which make it the best choice for MCU programming, these are the access to low-level capabilities like the ones found in Assembly language as well as supporting the powerful high-level functionalities. On the other hand, we have used Arduino libraries which are built on top of C++ programming language for implementing the gateway’s firmware; this was a safe choice for prototyping the network communication programming part. The software implementation of both the tracker and the gateway starts with testing the functionality of each hardware module based on circuits implemented in sections 3.2.5 and 3.2.6, after that, a complete firmware is written for both of the systems to perform the required task.

3.4.1. Tracker node firmware implementation

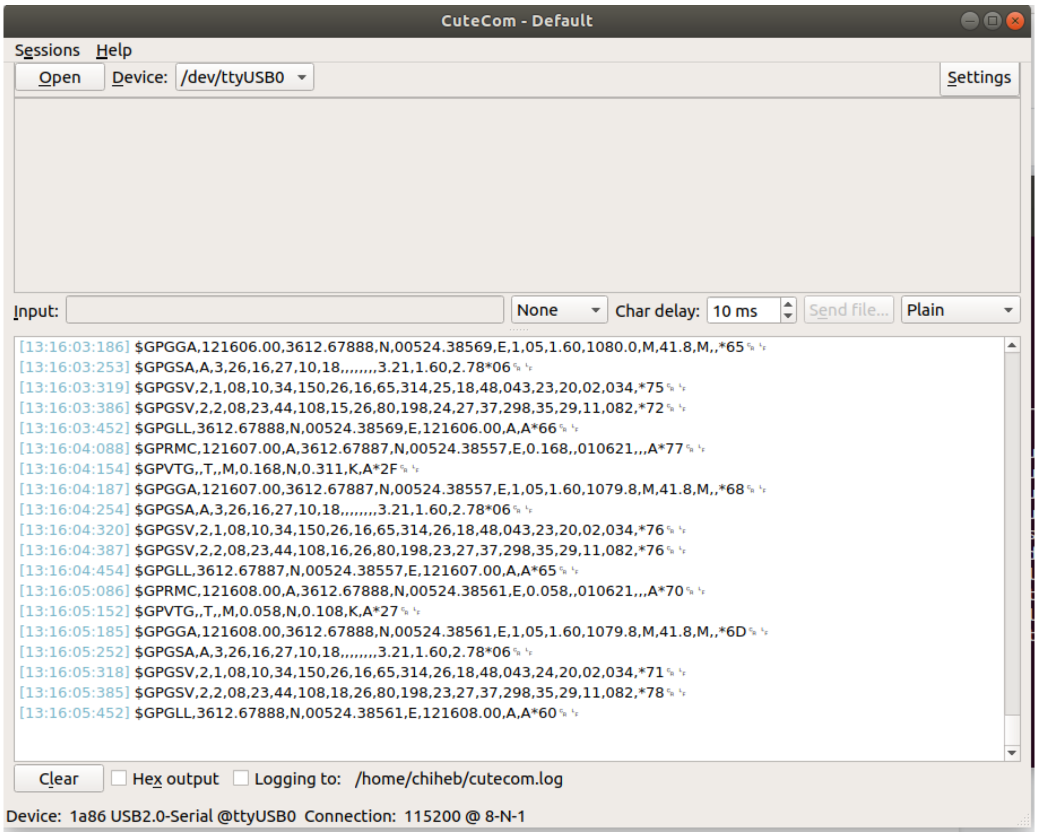
For the tracker node which was based on the ATmega328P MCU, see section 3.2.5, we have written a C-language firmware from scratch to drive the GPS and the LoRa modules. For that purpose, it was necessary to set up the AVR Toolchain which includes all required components for the firmware development process. The AVR Toolchain is a collection of tools/libraries used to create applications for AVR MCUs; this collection includes compiler, assembler, linker, and Standard C and math libraries.

1. *AVR-GCC*: AVR-GCC is a compiler that takes C language high level code and creates a binary source which can be uploaded into an AVR MCU. Thus AVR-GCC might be regarded as a 'C' cross compiler for producing AVR code.
2. *AVR-libc*: A subset of the standard C Library with some additional AVR specific functions. The libc-avr package includes C libraries, header files, and documentation primarily for the AVR target and is used in conjunction with AVR-GCC.
3. *AVRDUDE*: AVRDUDE is a utility to download/upload/manipulate the ROM and EEPROM contents of AVR MCUs using the in-system programming technique (ISP).

The development of the firmware was done on a Linux environment (Ubuntu) machine; we have used the terminal to install the AVR Toolchain before diving into the software implementation process. After installing all necessary components for the development process, we have tested first the functionality of each hardware module alone to make sure that we would not face any problem when building up the whole system. Below are the steps that we followed during the testing of the LoRa and the GPS modules:

1) *Testing the functionality of the NEO-6M*

As mentioned in the hardware implementation section 3.2.3, the NEO-6M U-blox GPS module uses UART to send GPS data to the MCU. The ATmega328P contains a built-in UART port that can be configured to send and receive data with different baud rates. First, we have initialized the UART port by setting the baud rate, enabling the transmitter and the receiver, and setting the data format that would be used for the communication, this was done by writing bits to specific locations on the configuration registers of the ATmega328P. Since the default baud rate of the NEO-6M is 9600 bits/sec, the same value was initialized at the MCU side. When testing the functionality of the GPS module, we had to configure the ATmega328P to receive the data from the GPS module then send it via UART as well to the host computer to observe the acquired data. The MCU can send data by writing a byte to the UART data buffer and receive data by reading it from the same buffer. After uploading the test firmware, we got a group of messages from the GPS module through UART, the messages were sent once every second meaning that the default transmission rate on the GPS module was one second as expected. Figure 3.18 shows the messages sent from the GPS module to the host machine via UART.



```
[13:16:03:186] $GPGGA,121606.00,3612.67888,N,00524.38569,E,1,05,1.60,1080.0,M,41.8,M,,*65
[13:16:03:253] $GPGSA,A,3,26,16,27,10,18,,,,,,,,3.21,1.60,2.78*06
[13:16:03:319] $GPGSV,2,1,08,10,34,150,26,16,65,314,25,18,48,043,23,20,02,034,*75
[13:16:03:386] $GPGSV,2,2,08,23,44,108,15,26,80,198,24,27,37,298,35,29,11,082,*72
[13:16:03:452] $GPGLL,3612.67888,N,00524.38569,E,121606.00,A,A*66
[13:16:04:088] $GPRMC,121607.00,A,3612.67887,N,00524.38557,E,0.168,,010621,,,A*77
[13:16:04:154] $GPVTG,,T,,M,0.168,N,0.311,K,A*2F
[13:16:04:187] $GPGGA,121607.00,3612.67887,N,00524.38557,E,1,05,1.60,1079.8,M,41.8,M,,*68
[13:16:04:254] $GPGSA,A,3,26,16,27,10,18,,,,,,,,3.21,1.60,2.78*06
[13:16:04:320] $GPGSV,2,1,08,10,34,150,26,16,65,314,26,18,48,043,23,20,02,034,*76
[13:16:04:387] $GPGSV,2,2,08,23,44,108,16,26,80,198,23,27,37,298,35,29,11,082,*76
[13:16:04:454] $GPGLL,3612.67887,N,00524.38557,E,121607.00,A,A*65
[13:16:05:086] $GPRMC,121608.00,A,3612.67888,N,00524.38561,E,0.058,,010621,,,A*70
[13:16:05:152] $GPVTG,,T,,M,0.058,N,0.108,K,A*27
[13:16:05:185] $GPGGA,121608.00,3612.67888,N,00524.38561,E,1,05,1.60,1079.8,M,41.8,M,,*6D
[13:16:05:252] $GPGSA,A,3,26,16,27,10,18,,,,,,,,3.21,1.60,2.78*06
[13:16:05:318] $GPGSV,2,1,08,10,34,150,26,16,65,314,26,18,48,043,24,20,02,034,*71
[13:16:05:385] $GPGSV,2,2,08,23,44,108,18,26,80,198,23,27,37,298,35,29,11,082,*78
[13:16:05:452] $GPGLL,3612.67888,N,00524.38561,E,121608.00,A,A*60
```

Device: 1a86 USB2.0-Serial @ttyUSB0 Connection: 115200 @ 8-N-1

Figure 3.18: GPS data received by the host computer via UART.

Figure 3.18 shows that the GPS messages sent via UART were in the NMEA standard format which was already discussed in section 2.1.1. NMEA format includes a set of messages that provides different GPS information; however, we were interested only in the GPS fix data that is the "GPGGA" field line. Our goal was to get the most important information out of this data line, which is the location data. The "GPGGA" message structure contains the following information:

```
$GPGGA,hhmmss.ss,Latitude,N,Longitude,E,FS,NoSV,HDOP,msl,m,Altref,m,Diff
Age,DiffStation*cs<CR><LF>
```

In our implementation, we have used three fields highlighted in green to get a valid location from the "GPGGA" message line:

1. *"FS" Position Fix Status Indicator field*: This field allows to determine whether a valid GPS location is fixed or not, the default value is 0, however, when the module fixes a position in regards to a number of satellites, this field changes to values 1 or 2 according to the number of satellites seen by the GPS receiver.

2. *Latitude and Longitude fields*: According to the NMEA Standard, Latitude and Longitude are output in the format degrees, minutes and fractions of minutes. To convert to degrees and fractions of degrees, the minutes and fractional minutes parts need to be converted. In other words, If the GPS receiver reports a Latitude of 4717.112671 and Longitude of 00833.914843 it means:
   Latitude: 47 degrees, 17.112671 minutes.
   Longitude: 8 degrees, 33.914843 minutes.
   Converted to degrees as follows:
   Latitude: (47 + 17.112671/60) = 47.28521118 degrees.
   Longitude: (8 + 33.914843/60) = 8.56524738 degrees.
   Degree is the format supported by the visualization tool that we would use later in the web application side.

After getting a clear idea about the required data fields that we had to use in order to detect a location, we could implement the first part of the firmware. Based on our software design, we had to implement an algorithm that reads GPS data from the module via UART, checks if the data is valid, and then extracts GPS latitude and longitude from the GPGGA message line. For that reason, we have built three functions to perform the required tasks; below is the pseudocode for the functions to check for valid GPS data, and extract the latitude and longitude from it:

```
GPS_valid( data[] ):
      read data from UART port
      if( received data == "$GPGGA" ):
            save GPGGA message line in data[]
            if( "FS" field in data[] == 1 or 2 ):
                  return true      // 1 or 2 means that GPS data is valid
            else
                  return false     // else means that GPS data is not valid

GPS_getLatitude( data[] )
      Latitude[], LatitudeDegree[], i
      for i from 0 to Latitude field index in data[]:
            Latitude[i] = data[i + Latitude field index]
      Convert Latitude[] to degrees and save in LatitudeDegree[]
      return LatitudeDegree[]

GPS_getLongitude( data[] )
      Longitude[], LongitudeDegree[], i
      for i from 0 to Longitude field index in data[]:
            Longitude[i] = data[i + Longitude field index]
      Convert Longitude[] to degrees and save in LongitudeDegree[]
      return LongitudeDegree[]
```

2) *Testing the functionality of the SX1278*

The SX1278 LoRa transceiver uses SPI communication protocol to send or receive data depending on the operation mode. As we are developing the firmware for the tracker node, the SX1278 acts as a transmitter on this side, however, we had to set up another LoRa module that worked as a receiver to be able to read the transmitted packets and to test both the functionality of the transmitter and the receiver once at a time. First, we have initialized the SPI port on the ATmega328P to work properly in regards to the SX1278 internal SPI port parameters. Similarly, to set up the SPI communication protocol on the ATmega328P, we have written bits to a number of configuration registers connected to the SPI port, the MCU was configured to work as a Master, the clock to drive the data exchange was set to 8Mhz, “CPOL” clock polarity set to low when idle, and “CPHA” clock phase set to sample on the leading clock edge; these parameters were chosen after checking the SX1278 SPI’s port default configuration from the datasheet. After setting up the SPI port on the MCU, it was possible to access internal registers of the SX1278; both read and write operations were allowed for all registers.

The LoRa module allows access to internal registers by receiving an address byte followed by a data byte for the write access whereas an address byte is received and a data byte is sent for the read access. Before each read or write operation, the NSS (negative slave select line) pin must be set to low and at the end of the operation, it must be returned to high. Below is the pseudocode of the functions to perform a read/write operation on the ATmega328P from/to the SX1278’s internal registers:

```
LoRa_writeRegister( address, value )
      set NSS to low  // select slave
      send address    // send the address of the register to read/write
      while( byte transmission is not done )
      send value      // write a data byte to the address register
      while( byte transmission is not done )
      set NSS to high // unselect slave

LoRa_readRegister( address )
      response        // variable to hold the response of the operation
      set NSS to low  // select slave
      send address    // send the address of the register to read/write
      while( byte transmission is not done )
      response = value read from the address register
      set NSS to high // unselect slave
      return response
```

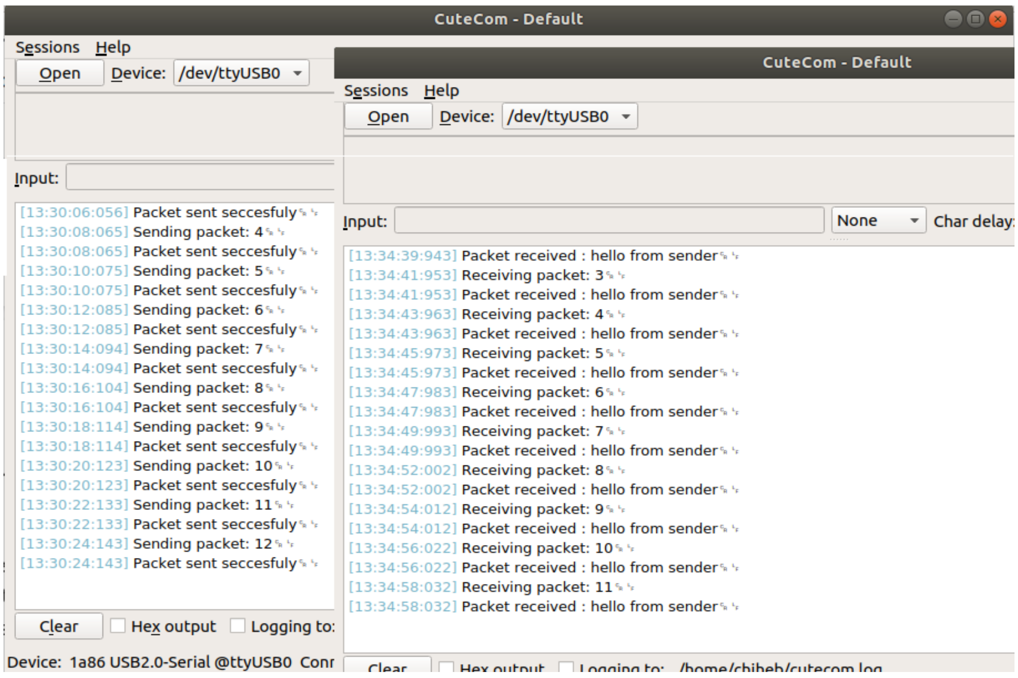
Before starting the wireless data transmission between the LoRa transmitter and receiver, it was necessary to configure both sides to use the LoRa communication protocol, to set the carrier frequency, to set up the base addresses of the FIFO register, to enable the CRC algorithm, to set the transmission power, and to put the modules on the standby mode. We had to write a function to initialize all parameters before starting the data transmission process, this function had to write different parameters to the module's internal registers using the *LoRa_writeRegister()* function described above. The following pseudocode presents the *LoRa_Begin()* function which initializes the LoRa module before any data communication operation can occur:

```
LoRa_Begin( frequency )
      set NSS to low                                    //select slave
      LoRa_writeRegister(mode_register, value)          //put in LoRa mode
      LoRa_writeRegister(frequency_register, frequency)//set frequency band
      LoRa_writeRegister(FIFO_base_register, value)//set FIFO base address
      LoRa_writeRegister(config_register, value)   //enable CRC
      LoRa_write(TX_power_register, power_value)   //set transmission power
      LoRa_writeRegister(mode_register, value)     //put in standby mode
      return true
```

The last two functions that we had to write in order to test the functionality of the SX1278 were responsible for handling the data sending and receiving processes. Similarly, *LoRa_send()* and *LoRa_receive()* functions, shown in pseudocode below, use the *LoRa_readRegister()* and *LoRa_writeRegisters()* to access internal registers to allow data transmission and reception. *LoRa_send()* initiates data transmission operation by writing a byte to the FIFO register of the LoRa module, whereas, *LoRa_read()* allows data reception by reading a byte from the FIFO register as well. Figure 3.19 shows packets sent from the transmitter side and received on the receiver side.

```
LoRa_send( data[], size )
      LoRa_writeRegister(mode_register, value)   // put in TX mode
      LoRa_writeRegister(FIFO_base_addr, address)// reset FIFO base address
      LoRa_writeRegister(packet_length_register, value)// set packet length
      for i from 0 to size:
            LoRa_writeRegister(FIFO_address, data[i] // send data
      LoRa_writeRegister(mode_register, value)       // put in standby mode

LoRa_receive( )
      LoRa_writeRegister(mode_register, value)       // put in RX mode
      LoRa_writeRegister(FIFO_address, LoRa_readRegister(current_RX_addr))
      LoRa_writeRegister(mode_register, value)       // put in standby mode
      return LoRa_readRegister(FIFO_address)
```



```
[13:30:06:056] Packet sent seccesfuly
[13:30:08:065] Sending packet: 4
[13:30:08:065] Packet sent seccesfuly
[13:30:10:075] Sending packet: 5
[13:30:10:075] Packet sent seccesfuly
[13:30:12:085] Sending packet: 6
[13:30:12:085] Packet sent seccesfuly
[13:30:14:094] Sending packet: 7
[13:30:14:094] Packet sent seccesfuly
[13:30:16:104] Sending packet: 8
[13:30:16:104] Packet sent seccesfuly
[13:30:18:114] Sending packet: 9
[13:30:18:114] Packet sent seccesfuly
[13:30:20:123] Sending packet: 10
[13:30:20:123] Packet sent seccesfuly
[13:30:22:133] Sending packet: 11
[13:30:22:133] Packet sent seccesfuly
[13:30:24:143] Sending packet: 12
[13:30:24:143] Packet sent seccesfuly
```

```
[13:34:39:943] Packet received : hello from sender
[13:34:41:953] Receiving packet: 3
[13:34:41:953] Packet received : hello from sender
[13:34:43:963] Receiving packet: 4
[13:34:43:963] Packet received : hello from sender
[13:34:45:973] Receiving packet: 5
[13:34:45:973] Packet received : hello from sender
[13:34:47:983] Receiving packet: 6
[13:34:47:983] Packet received : hello from sender
[13:34:49:993] Receiving packet: 7
[13:34:49:993] Packet received : hello from sender
[13:34:52:002] Receiving packet: 8
[13:34:52:002] Packet received : hello from sender
[13:34:54:012] Receiving packet: 9
[13:34:54:012] Packet received : hello from sender
[13:34:56:022] Receiving packet: 10
[13:34:56:022] Packet received : hello from sender
[13:34:58:032] Receiving packet: 11
[13:34:58:032] Packet received : hello from sender
```

Figure 3.19: Data communication between LoRa modules using the ATmega328P.

3) Implementing the firmware on the tracker node

At this point, and after testing the functionality of both the LoRa and GPS modules, we were able to write a complete firmware for the tracker node which was based on the functions that were written during the testing phase. The tracker firmware starts by initializing the UART and SPI protocols for data communication between the NEO-6M and the SX1278 modules. The LoRa module is then configured based on the required parameters set by the user; the firmware will break if any error occurs during the module's initialization. The tracker node then tries to get valid GPS data, extracts the latitude and longitude fields out of it, sends both information one at a time then waits for n seconds before receiving the next valid GPS location. The same operation is executed continually and infinitely as long as the core MCU has a power source attached to it. Below is the pseudocode for the tracker node system.

```
Begin
      data[], latitude[], longitude[]
      Initialize UART protocol
      Initialize SPI protocol
      if( !LoRa_begin() )
            return false
      while(true)
            if( GPS_valid( data[] ) )
                  latitude = GPS_getLatitude( data[] )
                  longitude = GPS_getLongitude( data[] )
                  LoRa_send( latitude )
                  LoRa_send( longitude )
                  delay( n_seconds )
End
```

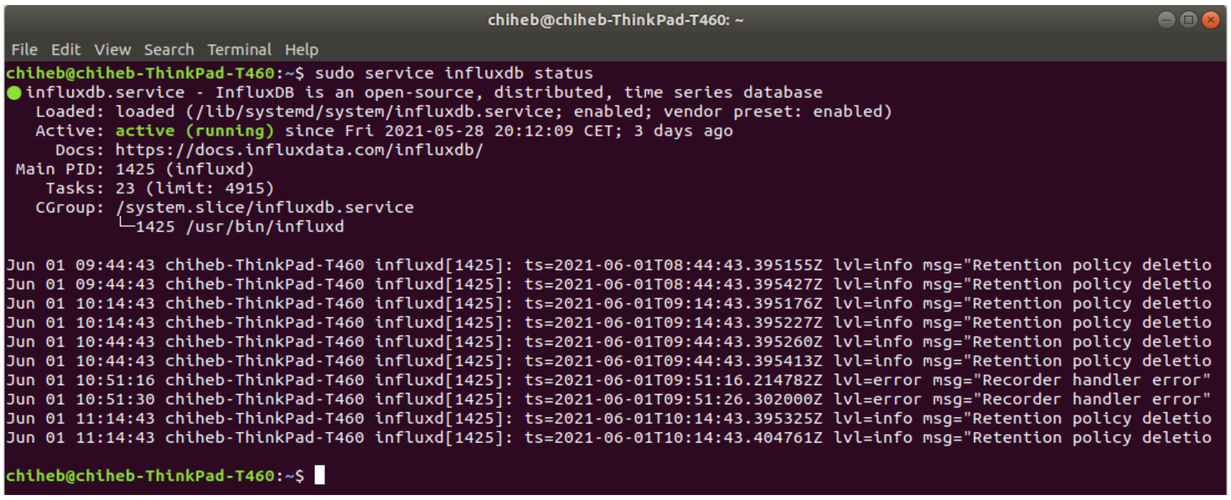
3.4.2. Web application installation

Before moving to the gateway software implementation, it was better to set up the web application first then test sending data from the gateway to the web application. By the web application we mean the data storage and the visualization tools; for that purpose we have used *InfluxDB* open-source time-series database for data storage and *Grafana* multi-platform open-source analytics and interactive web application for data visualization.

1) InfluxDB installation and testing

InfluxDB is a database management system developed by InfluxData, open-source, and can be used free of charge. In addition, it can run as a customizable local server with a web-based user interface that easily allows the insertion, deletion, and modification of data stored in the database. InfluxDB includes a built-in time service that uses the Network Time Protocol (NTP) to ensure that time is synchronized between all systems which makes it ideal for dealing with time-series data such as IoT data. On the other hand, we have used InfluxDB as it provides an Arduino client library to write data directly from the ESP8266 gateway into the database.

We have first installed the InfluxDB local server on our Linux host machine using commands provided in the influxDB official website. Figure 3.20 shows that InfluxDB local server was running properly. By default, InfluxDB uses TCP port "8086" for client-server communication; using a web browser, we could access the InfluxDB local server user interface. Figure 3.21 shows the welcome page of the influxDB local server user interface, at the leftmost of the page there exits different options to interact with the backend of the data management system such as inspecting data using data queries which can be executed directly on this web page.


chiheb@chiheb-ThinkPad-T460: ~
File Edit View Search Terminal Help

```
chiheb@chiheb-ThinkPad-T460:~$ sudo service influxdb status
● influxdb.service - InfluxDB is an open-source, distributed, time series database
   Loaded: loaded (/lib/systemd/system/influxdb.service; enabled; vendor preset: enabled)
   Active: active (running) since Fri 2021-05-28 20:12:09 CET; 3 days ago
     Docs: https://docs.influxdata.com/influxdb/
 Main PID: 1425 (influxd)
    Tasks: 23 (limit: 4915)
   CGroup: /system.slice/influxdb.service
           └─1425 /usr/bin/influxd

Jun 01 09:44:43 chiheb-ThinkPad-T460 influxd[1425]: ts=2021-06-01T08:44:43.395155Z lvl=info msg="Retention policy deletio
Jun 01 09:44:43 chiheb-ThinkPad-T460 influxd[1425]: ts=2021-06-01T08:44:43.395427Z lvl=info msg="Retention policy deletio
Jun 01 10:14:43 chiheb-ThinkPad-T460 influxd[1425]: ts=2021-06-01T09:14:43.395176Z lvl=info msg="Retention policy deletio
Jun 01 10:14:43 chiheb-ThinkPad-T460 influxd[1425]: ts=2021-06-01T09:14:43.395227Z lvl=info msg="Retention policy deletio
Jun 01 10:44:43 chiheb-ThinkPad-T460 influxd[1425]: ts=2021-06-01T09:44:43.395260Z lvl=info msg="Retention policy deletio
Jun 01 10:44:43 chiheb-ThinkPad-T460 influxd[1425]: ts=2021-06-01T09:44:43.395413Z lvl=info msg="Retention policy deletio
Jun 01 10:51:16 chiheb-ThinkPad-T460 influxd[1425]: ts=2021-06-01T09:51:16.214782Z lvl=error msg="Recorder handler error"
Jun 01 10:51:30 chiheb-ThinkPad-T460 influxd[1425]: ts=2021-06-01T09:51:26.302000Z lvl=error msg="Recorder handler error"
Jun 01 11:14:43 chiheb-ThinkPad-T460 influxd[1425]: ts=2021-06-01T10:14:43.395325Z lvl=info msg="Retention policy deletio
Jun 01 11:14:43 chiheb-ThinkPad-T460 influxd[1425]: ts=2021-06-01T10:14:43.404761Z lvl=info msg="Retention policy deletio

chiheb@chiheb-ThinkPad-T460:~$ █
```

Figure 3.20: InfluxDB local server status.

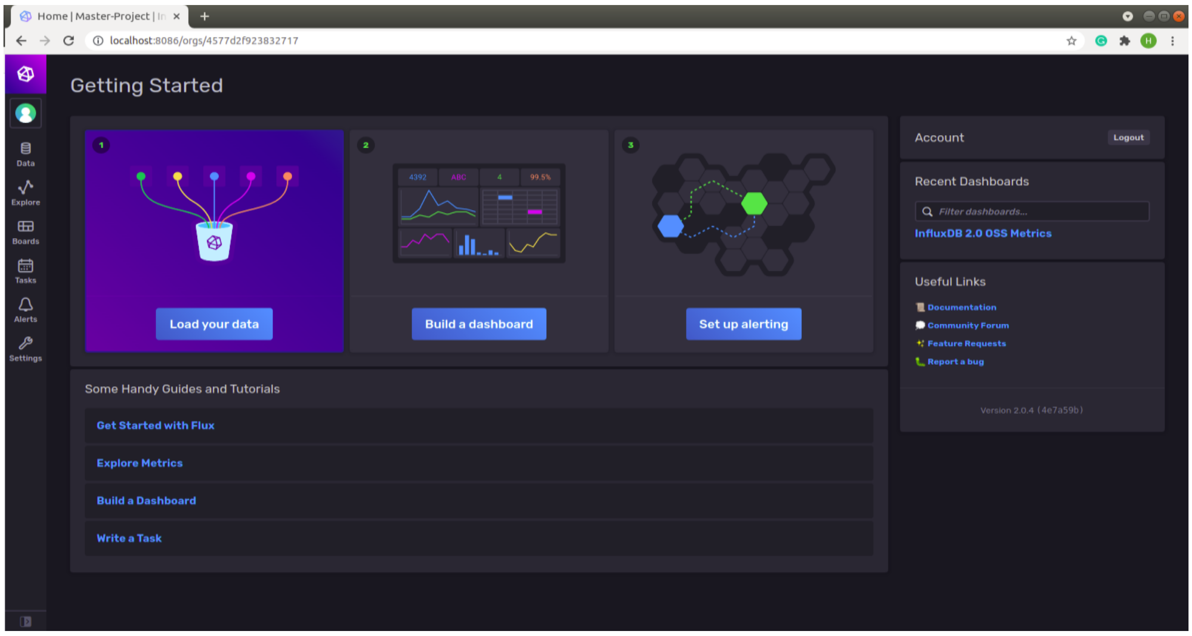
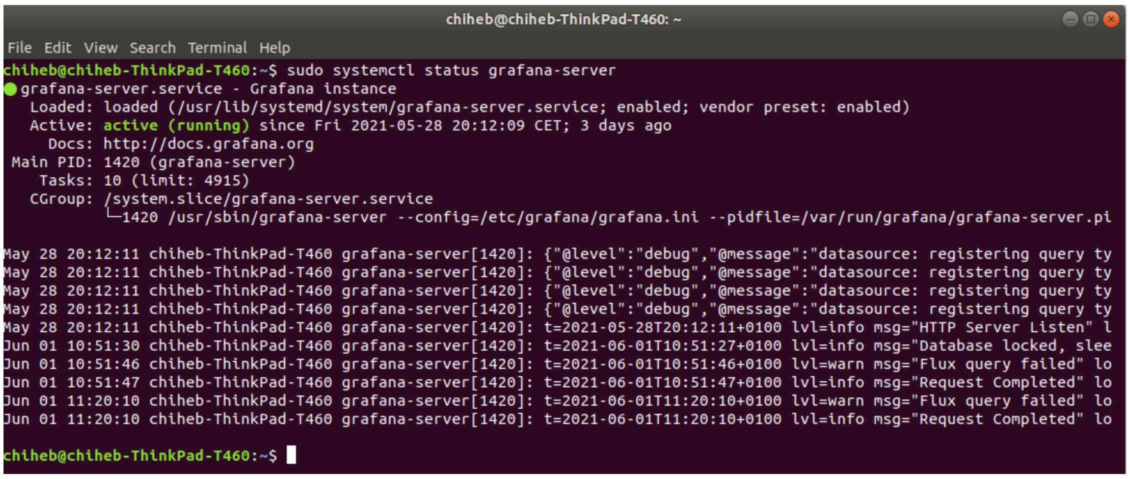


Figure 3.21: InfluxDB local server user interface.

2) *Grafana installation and testing*

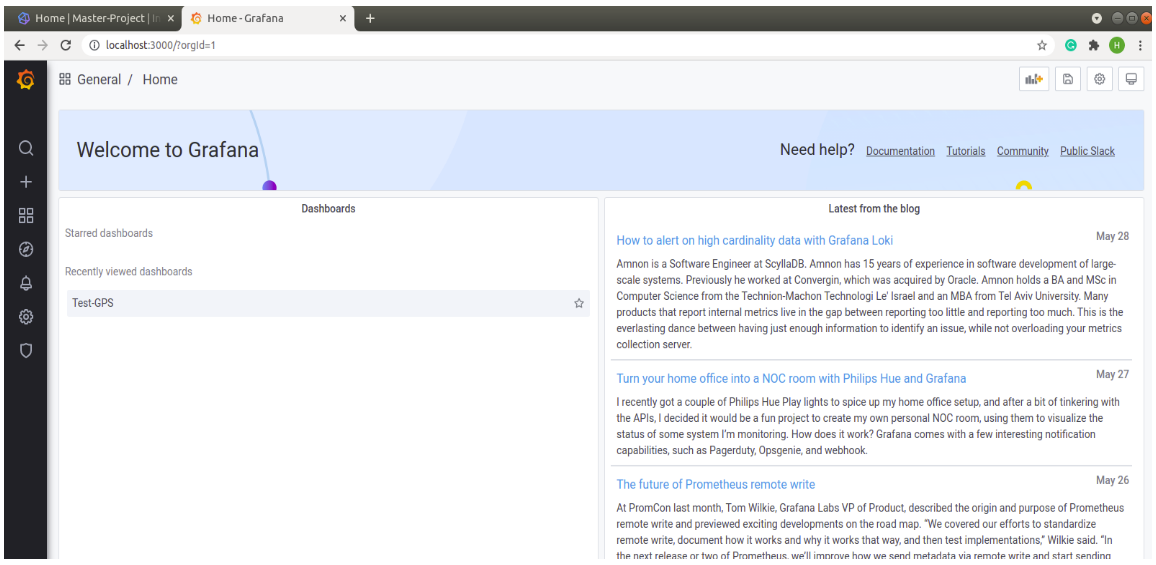
Grafana is an open-source visualization and analysis software that allows users to query, visualize, alert on, and explore metrics no matter where they are stored. Grafana provides tools to turn time-series database (TSDB) data into beautiful graphs and visualizations. We have chosen Grafana as a visualization tool as it allows data query from a wide range of data sources including influxDB database which is used for storage in our application. Grafana was installed locally using commands from the official website. Figure 3.22 shows that Grafana local server was running properly. By default, Grafana uses TCP port “3000” for client-server communication. Again, and using a web browser, we could access the Grafana local server user interface as shown in Figure 3.23. Grafana allows interacting with the backend server using the buttons on the leftmost of the user interface, they allow adding data sources, creating panels, and adding dashboards for visualization.


```
chiheb@chiheb-ThinkPad-T460: ~
File  Edit  View  Search  Terminal  Help
chiheb@chiheb-ThinkPad-T460:~$ sudo systemctl status grafana-server
● grafana-server.service - Grafana instance
   Loaded: loaded (/usr/lib/systemd/system/grafana-server.service; enabled; vendor preset: enabled)
   Active: active (running) since Fri 2021-05-28 20:12:09 CET; 3 days ago
     Docs: http://docs.grafana.org
 Main PID: 1420 (grafana-server)
    Tasks: 10 (limit: 4915)
   CGroup: /system.slice/grafana-server.service
           └─1420 /usr/sbin/grafana-server --config=/etc/grafana/grafana.ini --pidfile=/var/run/grafana/grafana-server.pi

May 28 20:12:11 chiheb-ThinkPad-T460 grafana-server[1420]: {"@level":"debug","@message":"datasource: registering query ty
May 28 20:12:11 chiheb-ThinkPad-T460 grafana-server[1420]: {"@level":"debug","@message":"datasource: registering query ty
May 28 20:12:11 chiheb-ThinkPad-T460 grafana-server[1420]: {"@level":"debug","@message":"datasource: registering query ty
May 28 20:12:11 chiheb-ThinkPad-T460 grafana-server[1420]: {"@level":"debug","@message":"datasource: registering query ty
May 28 20:12:11 chiheb-ThinkPad-T460 grafana-server[1420]: t=2021-05-28T20:12:11+0100 lvl=info msg="HTTP Server Listen" l
Jun 01 10:51:30 chiheb-ThinkPad-T460 grafana-server[1420]: t=2021-06-01T10:51:27+0100 lvl=info msg="Database locked, slee
Jun 01 10:51:46 chiheb-ThinkPad-T460 grafana-server[1420]: t=2021-06-01T10:51:46+0100 lvl=warn msg="Flux query failed" lo
Jun 01 10:51:47 chiheb-ThinkPad-T460 grafana-server[1420]: t=2021-06-01T10:51:47+0100 lvl=info msg="Request Completed" lo
Jun 01 11:20:10 chiheb-ThinkPad-T460 grafana-server[1420]: t=2021-06-01T11:20:10+0100 lvl=warn msg="Flux query failed" lo
Jun 01 11:20:10 chiheb-ThinkPad-T460 grafana-server[1420]: t=2021-06-01T11:20:10+0100 lvl=info msg="Request Completed" lo

chiheb@chiheb-ThinkPad-T460:~$ 
```

Figure 3.22: Grafana local server status.



How to alert on high cardinality data with Grafana Loki — May 28

Amnon is a Software Engineer at ScyllaDB. Amnon has 15 years of experience in software development of large-scale systems. Previously he worked at Convergin, which was acquired by Oracle. Amnon holds a BA and MSc in Computer Science from the Technion-Machon Technologi Le' Israel and an MBA from Tel Aviv University. Many products that report internal metrics live in the gap between reporting too little and reporting too much. This is the everlasting dance between having just enough information to identify an issue, while not overloading your metrics collection server.

Turn your home office into a NOC room with Philips Hue and Grafana — May 27

I recently got a couple of Philips Hue Play lights to spice up my home office setup, and after a bit of tinkering with the APIs, I decided it would be a fun project to create my own personal NOC room, using them to visualize the status of some system I'm monitoring. How does it work? Grafana comes with a few interesting notification capabilities, such as Pagerduty, Opsgenie, and webhook.

The future of Prometheus remote write — May 26

At PromCon last month, Tom Wilkie, Grafana Labs VP of Product, described the origin and purpose of Prometheus remote write and previewed exciting developments on the road map. "We covered our efforts to standardize remote write, document how it works and why it works that way, and then test implementations," Wilkie said. "In the next release or two of Prometheus, we'll improve how we send metadata via remote write and start sending

Figure 3.23: Grafana local server user interface.

3.4.3. Gateway firmware implementation

The gateway hardware implementation was based on the ESP8266 Wi-Fi module, see section 3.2.6. As is the case for all MCUs, ESP8266 allows different options for software development; however, in our case we have used the Arduino Integrated Development Environment which provides a full Toolchain to develop firmware for the ESP8266. Unlike the tracker node firmware which was implemented from scratch using C programming code, here we have used Arduino libraries which are built on top of the C++ programming language; this made it easier for us to develop the gateway's firmware and especially for implementing the network communication part as Arduino provides several libraries for all network communication interfaces; this was the main motivation to use Arduino IDE in this part of the implementation as if we tried to write a firmware from scratch, we would have to dive into network communication protocols and the way they work internally which was out of our subject mainly.

For the SX1278 LoRa module, we have used the Arduino-LoRa library from *sandeepmistry* which allows sending and receiving LoRa packets using the ESP8266 [38]. The library source code is based on the operation of reading from and writing to the SX1278 internal registers, similar to what has been implemented on the tracker node side .To test the functionality of the LoRa module with the ES8266, we have configured the SX1278 to work as a receiver while sending packets from the tracker node. The Arduino-LoRa library provides three functions to handle the receiving operation of packets:

- *LoRa_parsePacket()* returns the number of bytes received by the LoRa module.
- *LoRa_available()* returns True whenever the module finshes receiving the current byte.

- *LoRa_receive()* returns the received byte by reading the FIFO internal register.

After, uploading a test code to the ESP8266 we have received data packets from the tracker node which was running a test code as well. Figure 3.24 shows packets received by the ESP8266.

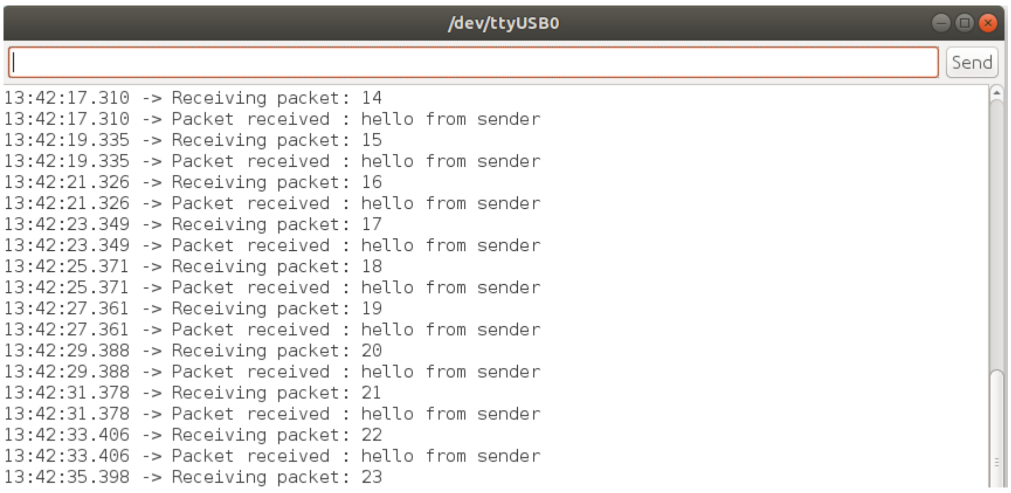
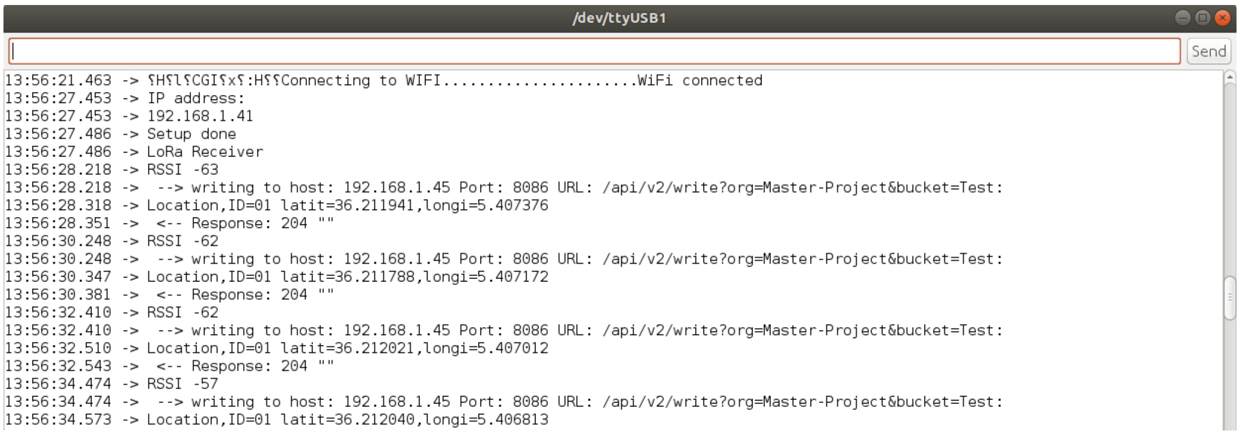

```
13:42:17.310 -> Receiving packet: 14
13:42:17.310 -> Packet received : hello from sender
13:42:19.335 -> Receiving packet: 15
13:42:19.335 -> Packet received : hello from sender
13:42:21.326 -> Receiving packet: 16
13:42:21.326 -> Packet received : hello from sender
13:42:23.349 -> Receiving packet: 17
13:42:23.349 -> Packet received : hello from sender
13:42:25.371 -> Receiving packet: 18
13:42:25.371 -> Packet received : hello from sender
13:42:27.361 -> Receiving packet: 19
13:42:27.361 -> Packet received : hello from sender
13:42:29.388 -> Receiving packet: 20
13:42:29.388 -> Packet received : hello from sender
13:42:31.378 -> Receiving packet: 21
13:42:31.378 -> Packet received : hello from sender
13:42:33.406 -> Receiving packet: 22
13:42:33.406 -> Packet received : hello from sender
13:42:35.398 -> Receiving packet: 23
```

Figure 3.24: LoRa packets received using the ESP8266.

Finally, we have used the ESP8266Wi-Fi library to connect to the Wi-Fi access point for data transmission over the network, and the InfluxDBV2 client library to write data into the database directly from the ESP8266 [39][40]. The gateway's firmware described below started by initializing the SPI communication protocol to be used for the SX1278, then to configure the LoRa module before any RF packets reception begins; again, if the LoRa module fails to configure, the system would break at this point of the operation. The next step was to connect the ESP8266 to a Wi-Fi access point which must be relatively near the gateway in order to have a stable network connection; this was done based on functions from the ESP8266Wi-Fi library. After setting up the parameters of the InfluxDB client which have the database name and the fields' names to be written to, we could initiate the data receiving operation. If the LoRa module captures any RF packet on the air, it will separate the latitude and longitude data from each other's, add tags that are necessary on the database side, and then forward the data to the web application over the network based on the InfluxDBV2 library functions. Figure 3.25 shows the ESP8266 connected successfully to the Wi-Fi access point and the data forwarded to the InfluxDB database as well.

```
Begin
      Inp[], latitude[], longitude[]
      Initialize SPI protocol
      if( !LoRa_begin() )
            return false
      connect to Wi-Fi
      Set database name                            // from influxDB
      Set columns names to be written to           // from influxDB
      while( true )
            if( LoRa_parsePacket() ):
                  while( LoRa_available() ):
                        Inp[i++] = LoRa_receive()
            // get latitude and longitude from the received packet
            for i from 0 to latitude field index:
                  latitude[i] = inp[i]
            for j from 0 to longitude field index:
                  longitude[i] = inp[i+ longitude field index]
            Add tags to latitude and longitude   // will be used in database
            Send latitude and longitude to database
End
```



```
/dev/ttyUSB1
Send
13:56:21.463 -> ʕHʕlʕCGIʕxʕ:Hʕʕ Connecting to WIFI.....................WiFi connected
13:56:27.453 -> IP address:
13:56:27.453 -> 192.168.1.41
13:56:27.486 -> Setup done
13:56:27.486 -> LoRa Receiver
13:56:28.218 -> RSSI -63
13:56:28.218 ->  --> writing to host: 192.168.1.45 Port: 8086 URL: /api/v2/write?org=Master-Project&bucket=Test:
13:56:28.318 -> Location,ID=01 latit=36.211941,longi=5.407376
13:56:28.351 ->  <-- Response: 204 ""
13:56:30.248 -> RSSI -62
13:56:30.248 ->  --> writing to host: 192.168.1.45 Port: 8086 URL: /api/v2/write?org=Master-Project&bucket=Test:
13:56:30.347 -> Location,ID=01 latit=36.211788,longi=5.407172
13:56:30.381 ->  <-- Response: 204 ""
13:56:32.410 -> RSSI -62
13:56:32.410 ->  --> writing to host: 192.168.1.45 Port: 8086 URL: /api/v2/write?org=Master-Project&bucket=Test:
13:56:32.510 -> Location,ID=01 latit=36.212021,longi=5.407012
13:56:32.543 ->  <-- Response: 204 ""
13:56:34.474 -> RSSI -57
13:56:34.474 ->  --> writing to host: 192.168.1.45 Port: 8086 URL: /api/v2/write?org=Master-Project&bucket=Test:
13:56:34.573 -> Location,ID=01 latit=36.212040,longi=5.406813
```

Figure 3.25: ESP8266 Wi-Fi connection and data transmission.

3.5. Results and discussion

To observe the full data path of our implementation, from the tracker node to the gateway until the visualization map, we have placed the tracker on a moving car while fixing the gateway in a building that has a Wi-Fi access point. The car was moving within a free range to test the reach of the LoRa module and the accuracy of the GPS module. Figure 3.26 shows a set of data points acquired during the car's movement, the yellow square represents the location of the gateway and the green circles show the car's locations during the movement. The data has been sent successfully from the tracking node, received by the gateway, forwarded to the influxDB database, and visualized as shown below in the Grafana visualization map; resulting the intended task that our solution was designed for, which is to track outdoor mobile vehicles. The green data points on the map give a clear idea about the current location of the asset, which allows the user to track the car in real-time and to collect all needed information about its location.

The gateway was put in three different floors inside of the building to check the LoRa signal range in regards to the height between the sender and the receiver. The gateway could receive data points from the tracker at a range of up to 293 meters. The red circle in Figure 3.26 was drawn on the map to demonstrate the part where the LoRa signal was missed, that part of the map contains a dense number of buildings which may act as an obstacle that prevents the RF signal from reaching the gateway. Placing the gateway in a higher location regarding the Lora nodes will increasingly allow greater reach for the LoRa signal. Table 3.1 shows different ranges acquired from our implementation during the test.

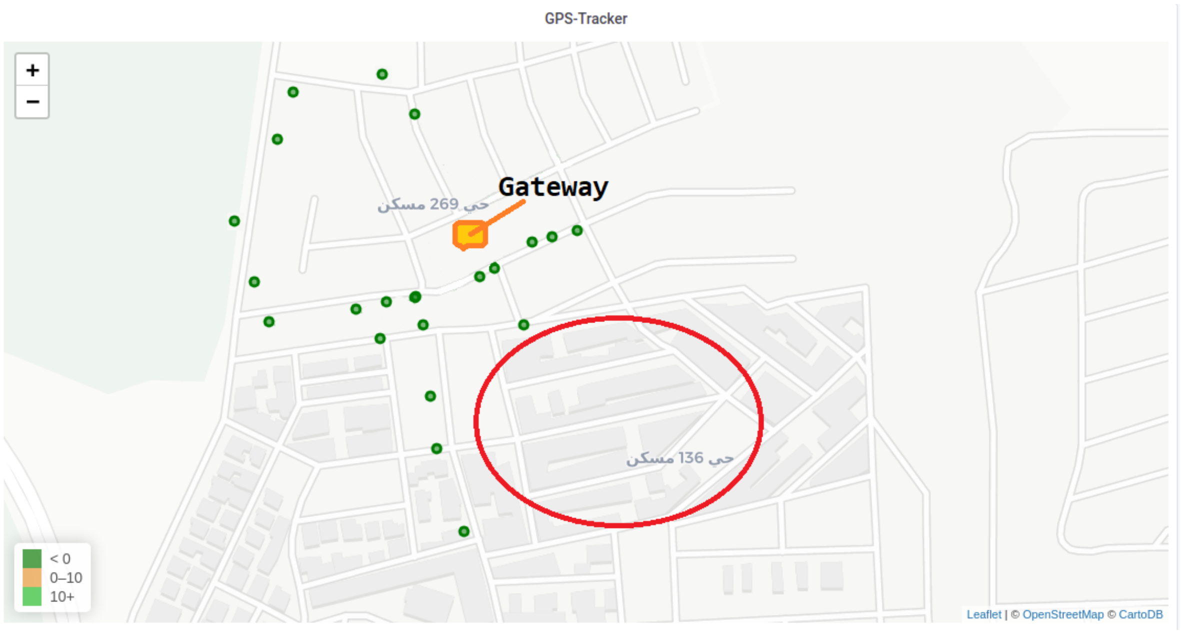


Figure 3.26: Tracking map test results from Grafana user interface.

Table 3.1: Gateway height vs LoRa range.

Gateway height (m)	Range (m)
2 (first floor)	<100
5 (second floor)	100-200
8 (third floor)	>200

During the test, the car was moving on the roads of the streets around the building where the gateway was fixed. It is clear that some of the points do not appear exactly within the road but next to it, this is due to the GPS module accuracy which is common when working with such technology. An error of 1-3 meters was observed about the location of the car. Figure 3.27 shows the real location of the car in the blue color and next to it the acquired location using our tracking node in the green color. Although there was an error in the data points on the map, this did not affect the information delivered by the tracker as it still gives the approximate location of an asset which is enough to locate the asset and perform a decision for such use case.

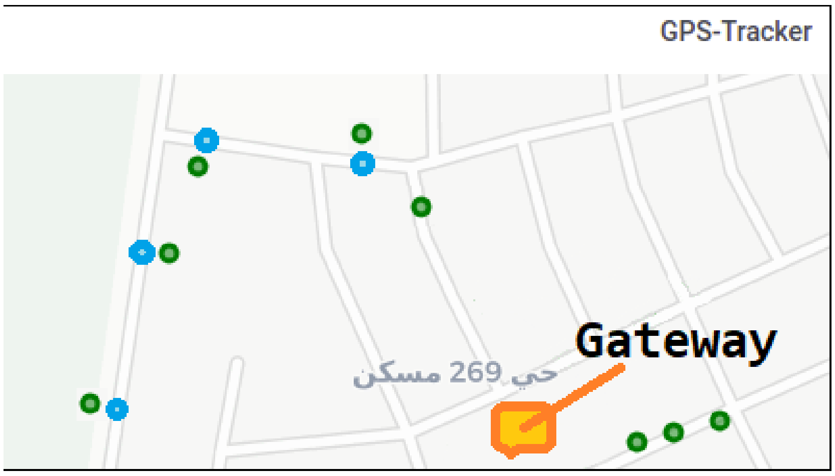


Figure 3.27: The acquired location vs the real location.

The firmware on the gateway was designed to capture the data only from a single node; however, it is possible to enhance the functionality of the gateway by implementing a real-time operating system-based firmware on top of more powerful computing unit to allow handling thousands of nodes simultaneously. On the tracker node side, we could implement logic to put the node on a sleep mode when the vehicle is not moving to increase the battery life while serving the intended operation. Regarding the GPS accuracy and the LoRa signal reach, the asset tracking solution based on these technologies works perfectly within large industrial environments, where tracking the location of constantly moving vehicles is greatly needed.

Conclusion

The IoT asset tracking solution was an attempt to deliver a cheap and reliable system that provides more meaningful insights to users interested in tracking assets within large industrial environments. Our design was based on GPS technology for localization and LoRa wireless communication technology for data transmission. The system consists of a gateway that receives location data a tracking node. The tracker node is the part responsible for localization and data transmission over long ranges. The gateway is a hub to collect data sent from nodes then forward these data to the web application where the storing and visualization occurs.

Both the tracker node and the gateway were implemented successfully using cheap hardware modules. First, the GPS and LoRa modules were tested individually to check the functionality of these parts; for each module, a software driver that runs on top of an MCU was written. Libraries were used in the gateway part to allow receiving RF packets along with the network interface data communication. The web application was installed and tested with the gateway as well. After setting up all parts of the solution, we could track a moving vehicle sending data wirelessly over approximately a long-range. Location data were received by the gateway then forwarded to the web application where data points appeared on the visualization map.

The design and implementation of the asset tracking solution prototype introduced a Proof-of-Concept that deals with a limited number of nodes, specifically, in our system the gateway could receive locations from one node only. For future work, an algorithm that handles thousands of nodes will be designed to enhance the functionality of the system to meet realistic use cases that are faced in the industry nowadays. Furthermore, the implemented system does not include an addressing logic when sending data from nodes to the gateway meaning that the gateway can capture any data on its range, for that reason, the LoRaWAN protocol can be integrated into our system as a full solution that provides nodes addressing, data encryption, a network server for management and much more.

Bibliography

[1] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of Things (IoT): A vision, architectural elements, and future directions. Future generation computer systems. Sep. 2013.

[2] I.F. Akyildiz, W. Su, and Y. Sankarasubramaniam, E. Cayirci, Wireless sensor networks: a survey. Computer Networks 38. March 2002.

[3] Tableau Software, Data visualization beginner's guide: a definition, examples, and learning resources. < https://www.tableau.com/learn/articles/data-visualization >. [Accessed Apr. 13, 2021].

[4] K. Ashton, That “Internet of Things” thing. RFiD Journal?. 2009.

[5] I. Lee, K. Lee, The Internet of Things (IoT): Applications, investments, and challenges for enterprises. Business Horizons. Jul. – Aug. 2015.

[6] Albert Behr, Best Uses of Wireless IoT Communication Technolog.< https://industrytoday.com/best-uses-of-wireless-iot-communication-technology/ >. Dec. 10, 2018. [Accessed Apr. 17, 2021].

[7] International Organization for Standardization, ISO 55000: Asset management — Overview, principles and terminology, Jan. 2014.

[8] Comparesoft Ltd, What is Asset Tracking, and How Does It Work?. < https://comparesoft.com/assets-tracking-software/what-is-assets-tracking/ >. [Accessed Apr. 19, 2021].

[9] L. Batistić, M. Tomic, Overview of Indoor Positioning System Technologies. MIPRO 2018. May 21-25, 2018.

[10] Brian Ray, Which Indoor Asset Tracking Technology is Best?. < https://www.airfinder.com/blog/which-indoor-asset-tracking-technology-is-best>. Apr. 26, 2019. [Accessed Jun. 25, 2021].

[11] Link Labs, Breaking it Down: The Building Blocks of Asset Tracking and Asset Monitoring. < https://www.link-labs.com/asset-tracking >. [Accessed Apr. 19, 2021].

[12] G. Lachapelle, GNSS Indoor Location Technologies. GNSS 2004, the International Symposium on GNSS/GPS. 2004.

[13] Official U.S. government, GPS: The Global Positioning System. < https://www.gps.gov/systems/gps/>. Feb 22, 2021. [Accessed Apr. 21, 2021].

[14] The National Marine Electronics Association. < https://www.nmea.org/ >. [Accessed Jun. 26, 2021].

[15] Eric Gakstatter , What Exactly Is GPS NMEA Data?. < https://www.gpsworld.com/what-exactly-is-gps-nmea-data/ >. Feb. 4, 2015. [Accessed Jun. 26, 2021].

[16] Trimble, NMEA-0183, Messages Guide for AgGPS Receivers. Feb. 2004.

[17] N.Rahemi, M. R. Mosavi, A. A. Abedi, and S.Mirzakuchaki, Accurate Solution of Navigation Equations in GPS Receivers for Very High Velocities Using Pseudorange Measurements, Department of Electrical Engineering, Iran University of Science and Technology, Narmak, Tehran 16846-13114, Iran. Jun. 29, 2014.

[18] Dali I., Mahbubur R., Abusayeed S., Low-Power Wide-Area Networks: Opportunities, Challenges, and Directions. The Workshop Program of the 19th International Conference. Jan. 2018.

[19] DP Acharjya and M Kalaiselvi Geetha, Internet of Things: Novel Advances and Envisioned Applications. 2017.

[20] Andrew J Viterbi, CDMA: principles of spread spectrum communication. Addison Wesley Longman Publishing Co., Inc. 1995.

[21] Semtech, AN1200.22, LoRa Modulation Basics. May, 2015.

[22] Semtech, What are LoRa and LoRaWAN?. < https://loradevelopers.semtech.com/library/tech-papers-and-guides/lora-and lorawan >. [Accessed Apr. 27, 2021].

[23] Semtech, What is LoRa.< https://www.semtech.com/lora >. [Accessed Apr. 27, 2021].

[24] IEEE Computer Society, IEEE Standard 802.15.4a-2007. New York, NY: IEEE. Aug. 31, 2007.

[25] Robert Lie, LoRa/LoRaWAN tutorial, < https://www.youtube.com/playlist?list=PLmL13yqb6OxdeOi97EvI8QeO8o-PqeQ0g >. [Accessed May 02, 2021].

[26] Martin Bor and Utz Roedig, LoRa Transmission Parameter Selection. 13th International Conference on Distributed Computing in Sensor Systems (DCOSS).2017.

[27] Manuel Jiménez, Rogelio Palomera , and Isidoro Couvertier , Introduction to Embedded Systems: Using Microcontrollers and the MSP430. 2014.

[28] Elecia White, Making embedded systems design patterns for great software. Nov 2011.

[29] Molly Galetto, What is Data Management?. < https://www.ngdata.com/what-is-data management/ >. March 31, 2016. [Accessed May 05, 2021].

[30] Oracle, what is a database. < https://www.oracle.com/database/what-is-database/ >. [Accessed May 05, 2021].

[31] InfluxData, what is a time series database. < https://www.influxdata.com/time-series-database/ >. [Accessed May 06, 2021].

[32] U-blox , U-blox 6 GPS Modules Data Sheet, Docu. No. GPS.G6-HW-09005-E.

[33] Semtech, SX1276-7-8-9 Datasheet.

[34] Atmel, Atmel ATmega328P Datasheet.

[35] Espressif, ESP8266EX Datasheet.

[36] Keith Shaw, Wi-Fi standards and speeds explained. < https://www.networkworld.com/article/3238664/80211x-wi-fi-standards-and-speeds explained.html >. Apr. 23, 2020. [Accessed May 14, 2021].

[37] Encyclopedia Britannica, <https://www.britannica.com/technology/TCP-IP >. [Accessed May 16, 2021].

[38] Sandeep Mistry, Arduino LoRa Library. < https://github.com/sandeepmistry/arduino-LoRa >. [Accessed Apr. 01, 2021].

[39] ESP8266 Community Forum, Arduino core for ESP8266 Wi-Fi chip, < https://github.com/esp8266/Arduino > [Accessed Apr. 02, 2021].

[40] Tobias Schürg, InfluxDB Arduino Client, < https://github.com/tobiasschuerg/InfluxDB-Client-for-Arduino >. [Accessed Apr. 03, 2021].